LASSE NIELSEN

REGULAR EXPRESSIONS AND MULTIPARTY SESSION TYPES WITH APPLICATIONS TO WORKFLOW BASED VERIFICATION OF USER INTERFACES

REGULAR EXPRESSIONS AND MULTIPARTY SESSION TYPES WITH APPLICATIONS TO WORKFLOW BASED VERIFICATION OF USER INTERFACES

LASSE NIELSEN

DEPARTMENT OF COMPUTER SCIENCE

FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Ph.D. thesis

Lasse Nielsen: *Regular expressions and multiparty session types with applications to workflow based verification of user interfaces,* Ph.D. thesis, © August 31, 2011

SUPERVISORS:
Fritz Henglein
Thomas Hildebrandt

LOCATION:
Copenhagen, Denmark

To the world and future generations in general, and my unborn son in particular.

ABSTRACT

With the aim of producing a framework for clinical practice guideline modelling and reasoning, we consider regular expressions and session types. We investigate and develop their ability to express example guidelines, verify treatment compliance, verify specification compliance and static verification of user interface implementations.

RESUMÉ

Under målsætningen at finde et framework til at modellere og resonere om retningslinjer for kliniske behandlinger, betragter vi regulære udtryk og session typer. Vi undersøger og videreudvikler deres egenskaber til at repræsentere eksempler på retningslinjer, kontrollere behandlinger, kontrollere behandlingsbeskrivelser samt statisk kontrol af brugerflade-implementationer.

PUBLICATIONS

Multiparty Symmetric Sum Types [96]
Presented at: EXPRESS 2010 – 17th International Workshop on Expressiveness in Concurrency
Included as Chapter 4

Regular Expression Containment:
Coinductive Axiomatization and Computational Interpretation [56]
Presented at: POPL 2011 – 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages
Included as Chapter 2

Bit-coded Regular Expression Parsing [94]
Presented at: LATA 2011 – 5th International Conference on Language and Automata, Theory and Applications

Bit-coded Regular Expression Parsing [95]
Submitted to: International Journal of Computer Mathematics
Included as Chapter 3

ACKNOWLEDGMENTS

I would like to use this opportunity to express my appreciation to the following people.

My advisors Fritz Henglein and Thomas Hildebrand for the guidance, cooperation and inspiration they have provided.

The Danish Strategic Research Agency, for funding the TrustCare project (Grant #2106-07-0019) which has enabled the presented work.

The people I visited during my stay abroad: Nobuko Yoshida, Kohei Honda and the mobility reading group of Imperial College and Queen Mary University of London, for being very welcoming and cooperative.

The (other) members of the APL group at DIKU who have provided the academic network and environment necessary to do scientific research. Special mentions goes to the lunch club.

Michael Nebel Nissen and Morten Ib Nielsen, who have been my project and study partners through university.

The department of Urology at Rigshospitalet who allowed me to observe their work. Special mentions goes to Line Lydom who helped arrange my visit.

Line Bie Pedersen and Jon Elverkilde who have chosen to write master theses related to my work.

Claus Brabrand and Jacob G. Thomsen who has shown interest in, implemented and given great feedback on parts of my work.

Finally my family and friends who have supported me and endured my negligence during this project. Special mentions goes to my loving partner Mette Hansen.

– Thank you!

CONTENTS

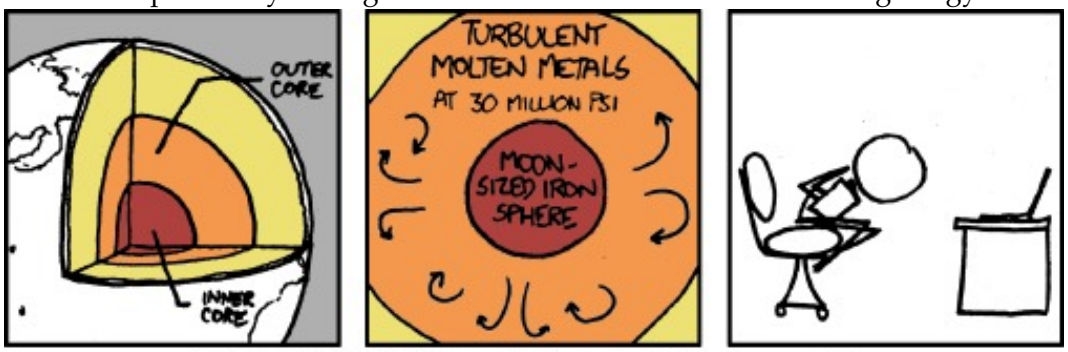
Part I

INTRODUCTION

1

INTRODUCTION

Thank you xkcd.com for many excellent comics.
This one captures my feelings about healthcare software as well as geology.



This introduction will underline the principal results, explain and demonstrate some applications of the work that we have done during this project. Parts of the results can be summarized in the following theses.

> THESIS 1: It is possible to produce compact representations of parse-trees efficiently, without explicitly materializing the represented parse-trees.

> THESIS 2: Multiparty asynchronous session types can be extended with some types of social interactions, such that workflows can be represented, and process compliance with the workflows can be verified via type-checking.

> THESIS 3: Programming languages can be integrated with formalised clinical practice guidelines in a way that allows guideline compliance verification of user interfaces by type checking.

OUTLINE

We will start by explaining the background and motivation for this project in Section 1.1. We then list the contributions of the project in Section 1.2. We introduce some existing workflow modelling frameworks in Section 1.3 before explaining how the contributions of the project can be applied to workflow modelling and reasoning in Section 1.4. In section 1.5 we report some of our observations from a field study. In Section 1.6 and 1.7 we list related work and ideas for future work, before concluding in Section 1.8.

1.1 BACKGROUND

A hospital is a hazardous place to be, not only because it is where you go when you are sick and thus in risk of getting worse, but also because the treatments you undergo can be dangerous or even deadly if the correct procedure is not followed very strictly. Examples of these dangers include administering medicine and performing surgery, and there is a wealth of possible errors spanning from simple technical errors such as misreading the prescribed dose of medicine to misdiagnosis from failure to notice or test for extra symptoms.

Adverse events

According to the danish patient-security database [3] there were 41501 reports of adverse events in Denmark during 2010. This number covers all events from hospitals, nursing homes etc. and includes some unavoidable events like when a patient falls over a rug. However 30% of the events occurred in the medication process, 17% were caused by interruptions and 15% were due to miscommunication, so many of these events could have been avoided.

The below examples illustrate that such errors occur. Specifically elements of a treatment can be overlooked, whether they be administration of medicine or investigation of symptoms.

Example 1 (Administration of antibiotics [111, Box 3, example 3])**.** *After an operation the patient must be administered antibiotics to avoid inflammation. In this case a woman suffered salpingitis, after an abortion where she did not receive antibiotics.*

Example 2 (Investigation of symptoms [111, Box 3, example 4])**.** *In this case a patient was treated and discharged. During the treatment, a symptom unrelated to the treated disorder was observed, but this was not investigated before discharging. Three months later the patient was readmitted, and had to undergo surgery to treat the cause of this symptom.*

Clinical Practice Guidelines

Errors are unavoidable, as all people make mistakes. There are of course many security measures to minimize the frequency of such errors, and clinical practice guidelines (CPGs) is one of the main approaches for this. CPGs are essentially

detailed descriptions of medical treatments. A CPG is produced by a group of doctors specialized in the concerned disorder, maintained regularly and explains how to diagnose and treat the disorder. They are used as guidelines for hospitals, to ensure that patients receive the same treatment at all hospitals, and that this is the best known treatment based on the current research and experience. Since a CPG is usually just a text document, it is hard to specify exactly what information is in a CPG. The content is very diverse, even when considering CPGs within a single database. Many CPGs include information about how to diagnose an illness, how to treat it, and a long appendix including the empirical evidence for the treatment described. A lot of the diagnose and treatment information in CPGs can be understood as workflow descriptions, and some CPGs even include summaries in the form of workflows [42], even though the form and content of these workflows is very diverse as well, spanning from checklist/timeline based explanations to complicated control-flow descriptions.

CPG adoption

As Example 1 and 2 shows, the CPGs are not always followed due to human mistakes, but another problem is to ensure that the correct CPG is used. New research or experiences means that sometimes the best treatment is revised. Either a problem is found with the used treatment, or a new and better treatment is discovered. In this case the CPG is updated, and the new practice should be adopted at the hospitals. The adoption of new CPGs happens by the doctors reading the new CPGs and manually adjust their treatments accordingly. This is a very liberal approach, which can result in slow adoption of new CPGs [45]. Some reasons for this may be that doctors are very busy, and have to prioritize treating patients over finding and reading CPG updates, and some doctors choose to use the original treatments because they do not believe in the newest treatments. This defeats the purpose of CPGs, because patients will receive different treatments depending on which doctor they are assigned.

Paper forms and checklists

Most treatments are currently conducted involving paper forms and checklists which are added to the patients journal. The forms and checklists can be viewed as derivatives of the CPGs, where filling out the forms requires the tests required by the CPGs to be performed, and the checklists include the steps in the CPGs to ensure that no steps are missed. In this way the forms and checklists are means to ensure that the CPGs are followed. The forms and checklists allow the treatments to be conducted without too much overhead, and they do help to avoid many mistakes. A good example of this was the introduction of a specific surgical checklist, where a large WHO study estimated the impact to decrease the deathrate of the observed surgeries by up to 50%, and inpatient complications by 35% [53].

These results motivate the use of tools such as paper forms and checklists to cultivate the adherence to CPGs. There are however limits to what can be obtained using paper forms and checklists. For example it is not possible for a paper form to express that one action excludes another action, which is often the case in

CPGs. These limits are due to the simple nature of paper forms and checklists, and it is necessary to use more expressive tools in order to represent and cultivate these types of workflow constraints. One way to cultivate more complex workflow constraints in treatments is to use software systems in stead of or in addition to the paper forms and checklists.

Healthcare Software

Over the recent years software has been introduced to hospitals in many ways. The complex instruments need software to function, and software systems have been used to improve the efficiency of many technical tasks such as operation scheduling, dictation and exchange of testresults between different wards. Software systems are also used to help avoid some of the simple technical errors in the treatment processes. Examples of such systems include medication prescription and administration systems, electronic patient journals and decision support systems. These systems are also partially designed from the CPGs. Such software systems should be used with caution, because the quality of the used software is at least as critical as the tasks it is used to perform. This is illustrated by the examples below.

Example 3 (Therac-25 software error [82])**.** *The Therac-25 was a radiation therapy machine, where software errors caused the machine to malfunction and expose the patient to a beam of radiation so intense, that it caused the tissue to wither away resulting in multiple deaths and permanent disabilities. It took years to discover that the software was to blame. When the software was analysed, the cause of the malfunctions was discovered to be two different software errors. One was an inconsistent update of the state, and the other was a byte-overflow error.*

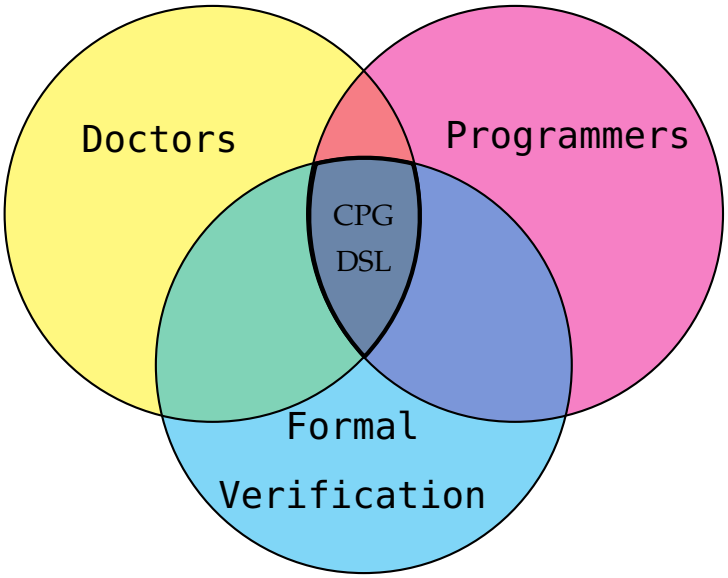
Example 4 (Electric turning aid [5])**.** *In a nursing home, a patient is placed in bed on a disabled electric turning aid. After some time the patient is found facing the mattress. The patients is blue in the face and breaths weakly. The patient is treated with oxygen, and regains conciousness. Due to a technical error the electric turning aid had started by itself, and caused the patient to be turned around.*

Example 5 (Software crash [5])**.** *In this situation, a patient in a nursing home needs to be hospitalized quickly. The patients data should be printed and sent with the patient, but the electronic patient journal crashes displaying error messages, so only parts of the data is successfully printed and sent with the patient.*

The danish department of healthscience (DSI) investigation of adverse events *from 2010 [70] mentions that software crashes increase the risk of adverse events. It also includes several comments from the questioned staff, expressing that such crashes occur frequently in practice.*

Therefore it is very important that software systems for the healthcare sector is of high quality, and in particular they should comply with the CPGs. This raises the question of what it means for a software system to comply with a CPG. If the system performs actions autonomously, then these actions should of course

Figure 1 Illustration of CPG language constraints



be allowed by the CPG. However most software systems for the healthcare sector does not perform actions autonomously, but allows the users to register or perform actions using user interfaces (UIs). In this case compliance means that the actions that can be registered and performed using the UI must be allowed by the CPG, and this is the compliance we have set out to formalize and verify.

Constraints for a CPG DSL

There is a communication problem, when developing and testing software systems that must comply with CPGs. The software is written by programmers, and not doctors and domain specialists who can better understand the CPGs. This raises the question if the produced systems actually comply with the CPGs they were meant to, because even if the system is well written and tested, the programmers and testers may have misread the CPGs. In order for the communication to succeed, the CPGs must be written in a domain specific language (DSL) that both the doctors and programmers understand.

Even though the programmer understands the CPGs, there is the possibility of bugs as in all software, that may cause the software to violate the CPGs. In the healthcare area the consequences of such bugs may be dire, as observed in Example 3, 4 and 5. One way to ensure that the guidelines are followed, is by using computer interpretable guidelines (CIGs). With CIGs the guidelines are written in a DSL which can be interpreted directly by a computer. This means that it is not necessary to write the software systems to use, as the guidelines provide these systems directly by interpretation. This requires the guideline description to contain all the information used in a software system, which may cause the

CIGs

Compliance verification

guideline language to be cumbersome, or the UIs to be too simplistic and limited.

Another way to ensure that the written software will not violate the CPGs is to formally verify the CPG compliance of the software. This requires the CPG DSL to be understandable by both doctors and programmers, and have a formal semantics that can be used to verify UI implementations, as illustrated in Figure 1. Such a language may be too restrictive to describe all the details of a CPG, but the aspects that can be expressed in the formal language, could then be ensured in the software systems. Verifying software manually is a tedious and error prone task, so in order for the verification to be practically applicable, it is often necessary to make the verification automatic.

Using automatic verification of CPG compliance, the problems with the expressiveness of paper forms, the communication between doctors and programmers and the adoption of new CPGs can be reduced. It is our hope, that the verification of software systems for the healthcare sector to comply with the CPGs can bring a whole new level of reliability and maintainability to healthcare software. This would mean that software systems could finally be trusted to cultivate the compliance with CPGs in the same way as paper forms and checklists are used today, enabling a much richer level of details from the CPGs. This would mean a partial realization of trustworthy software for the healthcare sector.

1.2 CONTRIBUTIONS

We will now briefly summarize the main results of our work. The results below have been divided into the three categories: regular expressions, session types and workflow based verification.

1.2.1 *Regular expressions*

In Chapter 2 we have observed the direct correspondence between the values of regular expressions as types and parse trees for regular expressions as grammars. This is not explicitly pointed out by neither the works on regular expressions as types [46] nor the works on regular expressions as grammars [22, 36].

In Chapter 2 we have developed a compact bit coding of parse trees for regular expressions, that allows efficient processing, by applying the oracle based coding of proofs [89] to proofs of string membership of regular expressions.

In Chapter 3 we have used the efficient representation of parse trees to simplify and optimize the known regular expression parsing methods, which allows us to generate the bit coded parse-trees efficiently without explicitly materializing the represented parse-trees.

In Chapter 3 we have implemented the optimized regular expression parsing algorithms, and used the implementations to compare the efficiency of the different parsing methods.

In Chapter 3 we have adapted Mohris [87] algorithm for minimizing sequential transducers, to minimize the generated output-deterministic transducers, and captured the effects of the prefix form step of the algorithm semantically.

In Chapter 3 we have implemented the transducer minimization, and used it to measure the effect on the generated transducers and enhanced DFAs, measure the effect of the prefix form step and to optimize the memory footprint and runtime of the transducer based regular expression parser algorithms.

In Chapter 2 we have created a new axiomatization of language inclusion for regular expressions. The new axiomatization is based on a general coinduction rule, and has an operational interpretation as coercions translating parse-trees for one regular expression to parse-trees of the other regular expression.

In Chapter 2 we have formalized the relation between coercion totality and axiomatization soundness, and created two syntactic conditions that ensure coercion totality and thus soundness of the axiomatization.

In Chapter 2 we have encoded the existing axiomatizations of language containment, which provides an operational interpretation to these axiomatization, and reveals the difference in how the axiomatizations ensure soundness.

1.2.2 *Session types*

In Chapter 4 we have observed that some workflow constructs cannot be represented naturally in the multiparty asynchronous session types [61].

In Chapter 4 we have extended the multiparty asynchronous session types with a simple type of social interaction called symmetric sum types, in a way that preserves subject reduction, and proved by example that workflows can be represented in a natural way using the extension.

In Chapter 4 we have studied the extensions effects to the expressiveness of well type terms by encoding well typed processes in the extended type-system as well typed processes in the original type-system, and proved semantic soundness and completeness of the encoding, despite the fact that the encoding does not enjoy full abstraction.

In Chapter 4 we have adapted Gorlas [51] encodability criteria to the setting of typed processes, and proved our encoding fulfills the adapted criteria.

In Chapter 5 we have merged the symmetric sum types extension with another

extension of multiparty asynchronous session types called assertions [20], such that the type-system still ensures subject reduction.

We have implemented an interpretor for the asynchronous π-calculus with multiparty sessions, symmetric synchronization and a type-checker for multiparty asynchronous session types with symmetric sum types and assertions [1].

1.2.3 *Workflow based verification*

In Section 1.4.1 we have represented a real world CPG as a regular expressions, which allows treatment verification via regular expression based matching, and guideline verification via language inclusion.

In Section 1.4.2 and 1.5 we have represented real world CPGs as multiparty asynchronous session types, which enables verification of automated participants implemented in the asynchronous π-calculus with symmetric synchronization via type-checking.

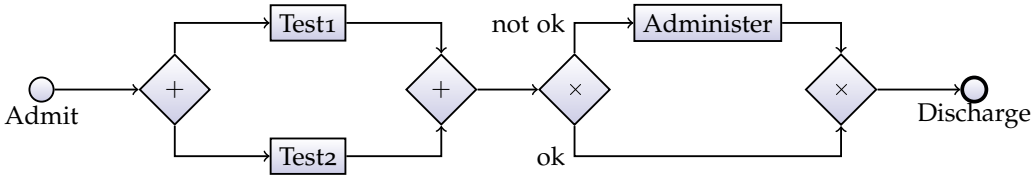
We have added and implemented UI primitives to the asynchronous π-calculus with multiparty sessions, symmetric synchronization and assertions [1], which enables the implementation of UIs in the asynchronous π-calculus, and verification of the implemented UIs compliance with CPG workflows represented as multiparty asynchronous session types with symmetric sum types and assertions via type-checking.

In Appendix A.2.2 and A.2.3 we have implemented the UIs for the modelled CPG workflows and verified that they comply with the represented workflows via type-checking. This is to our knowledge the first time UI implementations have been statically verified to comply with real world CPG workflows via type checking.

1.3 EXISTING WORKFLOW MODELLING FRAMEWORKS

Before we describe the applications of our contributions, we introduce some of the many related frameworks. This is to provide the reader with a description of the *state of the art* in the field, but also because some of the examples and workflow descriptions we use require a basic understanding of some of these frameworks, in particular the Process Matrix. For each of the selected models, we will give a short description of the models and show the representation of an example workflow. The example workflow we will use is very simple, and represents a standard treatment paradigm. The described workflow is activated, when a patient is admitted. First two tests are executed, possibly in parallel. Then, depending on the result of the tests, either the patient is discharged directly, or the patient is treated before discharging. In this workflow the treatment consists of administering

Figure 2 Example BPMN workflow



a drug to the patient. The workflow is ended, when the patient is discharged.

CIGs

There are many workflow based software frameworks, which offer some of the properties that we are interested in. The systems that offer UIs are based on a formalism of computer interpretable guidelines (CIGs), which is a formal language used to define workflows, such that computers can interpret the workflows and the performed actions to aid the further treatment. This requires the UI design and implementation to be included in the model.

BPMN and BPEL

The best known framework is probably the business process modelling notation (BPMN) [124]. BPMN is a graphical notation similar to flowcharts for describing business processes, and there are tools for creating and executing BPML diagrams [67], by translating the BPMN model to the business process execution language (BPEL) [9]. A BPEL program consists of a set of interacting web services. The recently released BPMN version 2 [6] includes more advanced interaction specifications such as choreographies which can be seen as binary session types without delegation. Since business processes are basically workflows similar to CPGs, BPMN is also useful for CPG modelling, and allows the modelled CPGs to be executed. There is however no guarantee that the specified processes are deadlock free or have communication safety. The BPMN model of the example workflow is in Figure 2.

UML activity diagrams

UML activity diagrams (UML-ADs) [106] is a graphical representation of workflows similar to flowcharts. There are no tools for executing models represented as UML-ADs, but there is a petri-net based standard semantics, so it should be possible to interpret UML-AD guidelines in the same way as the other guideline

Figure 3 Example UML-AD workflow

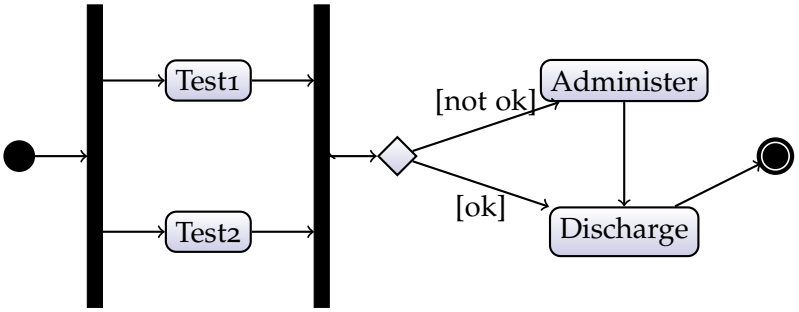
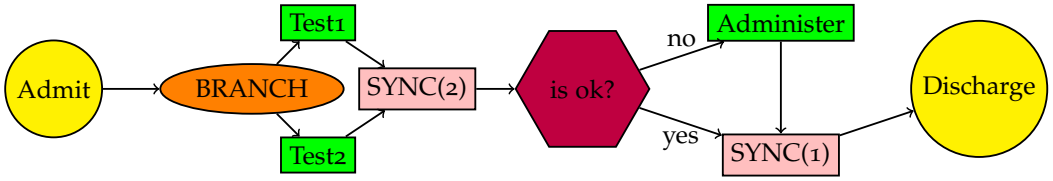


Figure 4 Example GLIF workflow



representations. The UML-AD model of the example workflow is in Figure 3.

The guideline interchange format

The guideline interchange format (GLIF) [97] is a modelling framework with three levels of abstractions. At the *conceptual level*, a model is simply a flowchart, describing the control flow of the modelled treatment. At the *computable level* – also known as the GLIF model – the workflow is represented in an object oriented structure, where each guideline, action, criterion and piece of data is represented by an object of the respective class. The *implementable level* – also known as the GLIF syntax – can be used to provide additional information to the models. For example the structure of the data can be specified, which helps the integration between systems that use the same or overlapping guidelines. GLIF models can be executed using a guideline execution engine [122]. The GLIF model of the example workflow is in Figure 4.

Process matrix

The process matrix [85] is a representation of workflows as a table, with one row for each action. Each action can be performed multiple times in the same execution of a workflow, and the workflow terminates when all the actions are in an executed (or inactive) state. The process matrix has some fixed columns and one extra column for each participant role. The fixed columns describe action information such as its name, but also sequential and logical precedence between the actions. If for example an action B has another action A as a sequential predecessor, then B cannot be executed unless A has already been performed (is in an executed state). Similarly, if B has A as a logical predecessor, then B cannot be executed unless A has already been executed, but additionally B is reset when A is performed (set to not executed), and thus B must be re-executed after A in order to complete

Figure 5 Example process matrix workflow

ID	Name	···	Seq	Log	Condition	Input
1.1	**Tests**					
1.1.1	Test1	···				result1
1.1.2	Test2	···				result2
1.2	**Treatment**		1.1			
1.2.1	Administer	···			result1 $\vee$ result2	
1.3	**Discharging**		1.2			
1.3.1	Discharge	···				

the workflow. Each action can also have an activity condition, which is a boolean expression that determines if the activity should be included in the workflow. When the activity condition of an action is false (the action is inactive) then the activity is ignored in the workflow until the state is changed in a way that causes the activity condition to become true again. The column for each participant describes the permissions of the participant, if the cell in the row of an action contains an N, then that user has no permissions for that action, meaning he cannot execute the action, and cannot read the data entered when other participants executes it. If it contains an R, then the user has read permissions. If it contains a W then the user has read permissions and can execute the action. Finally, the actions can be arranged in groups, which can be considered as stages in the workflow. As soon as all the actions in a group are in an executed (or inactive) state, the group is terminated, which means that none of the actions in the group can be executed again.

All the other CIGs investigated represent workflows in a way similar to flowcharts, and thus the process matrix stands out, because concurrent execution of actions does not have to be explicitly allowed. In stead concurrency must be explicitly limited using the sequential and logical predecessor constraints and activity conditions. The process matrix for the example workflow is in Figure 5.

Special purpose systems

Besides the CIGs there is a myriad of special purpose systems, offering decission support systems [44], real time reminders [65], medication prescription and administration management, electronic patient journals, operation scheduling and so on, but these are not as related to our work as CIGs.

1.4 REGULAR EXPRESSIONS AND SESSION TYPES AS WORKFLOW MODELS

We will now explain how the studied frameworks can be used as DSLs to express CPG workflows and work with CPGs formally.

1.4.1 *Regular expressions*

It may be surprising to some people that we have studied regular expressions [33] as a workflow model, because they are mostly used for string matching. It is however a kleene algebra over sequences of events, and the application to string matching is obtained by restricting the considered events to chars. If we do not assume that the events are chars, regular expressions can be used to express regular sets of sequences of events in a simple and intuitive way. Since CPG workflows essentially describe the allowed sequences of events in a treatment, regular expressions enable a simple and intuitive representation of some CPGs, and the development of regular expression theory and reasoning allows the same reasoning about the workflows represented in this way.

Representation of CPGs

Figure 6 Actions and participants in the oncology administration workflow

Action	Abbreviation	Participant	Abbreviation
Register basic patient information	bi	Doctor	D
Register lab results	lr	First nurse	N1
Register patient history	ph	Second nurse	N2
Calculate dose of chemotherapy	cd	Controlling pharmacist	CP
Sign order calculation	so	Pharmacist assistant	PA
Verify order	vo		
Reject order	ro		
Make preparation	mp		
Sign preparation	sp		
Verify preparation	vp		
Reject preparation	rp		
Checkout preparation	cp		
Check order and preparation match	cm		
Sign administration	sa		
Administer the preparation	ad		

Regular expressions are generally well know, and are formally defined in Part ii. Please note that we use the operator + for alternation as opposed to 1 or more iterations which is the normal interpretation.

In order to illustrate how CPGs can be represented by regular expressions, we

Figure 7 Regular expression representing the oncology administration workflow

```
// Register Patient Info
(D+N1)bi (D+N1)lr (D+N1)ph
// Order and Prepare
(Dcd Dso CPro)*              // Rejected orders
Dcd Dso (CPvo + CPro Dvo)    // Verified order
(PAmp PAsp CPrp)*            // Rejected preparations
PAmp PAsp (CPvp + CPrp PAvp) // Verified preparation
CPcp(N1cm N2cm + N2cm N1cm)  // Check correct patient
(D+N1)sa                     // Approve administration
// Administer Medicine
(N1+N2)ad
```

Figure 8 Regular expression representation including permutations

```
// Register Patient Info
(D+N1)bi (D+N1)bi* ((D+N1)lr ((D+N1)bi+(D+N1)lr)* (D+N1)ph+
                    (D+N1)ph ((D+N1)bi+(D+N1)ph)* (D+N1)lr)+
(D+N1)lr (D+N1)lr* ((D+N1)bi ((D+N1)bi+(D+N1)lr)* (D+N1)ph+
                    (D+N1)ph ((D+N1)lr+(D+N1)ph)* (D+N1)bi)+
(D+N1)ph (D+N1)ph* ((D+N1)bi ((D+N1)bi+(D+N1)ph)* (D+N1)lr+
                    (D+N1)lr ((D+N1)lr+(D+N1)ph)* (D+N1)bi))
```

give a regular expression for the real workflow used to administer chemotherapy [84].

Example 6 (Administration of chemotherapy [58, 84])**.** *This workflow has five participants: a doctor, two nurses, a controlling pharmacist and a pharmacist assistant, not counting the patient who is obviously a participant but not included because he does not perform any actions. The workflow has three stages. First the patient information is registered. Then the chemotherapy medicine dose is calculated, verified, prescribed, prepared and the preparation is verified. Finally the chemotherapy medicine is administered to the patient. The paper describing the workflow [84] uses the process matrix included in Appendix A.1.1 to specify the workflow. We have added an extra column to the matrix, where the input for each action is specified.*

The first thing to consider when representing a workflow is what events can occur, and how to represent them. An event in this workflow consists of a participant performing an action. The participants and possible actions are given in the original paper, but we list them in Figure 6 including abbreviations. We will use the abbreviations for the participants and actions as Σ*, and a workflow event is expressed as the participant event followed by the action event. For example the event "*The first nurse administers the preparation*" would be represented by the two atoms* `N1ad`.

Now that we can represent each event, the workflow can be represented by the regular expression in Figure 7, although we have cheated a little because the actions for registering the patient information can be permuted freely, and repeated until all the information has been registered. In order to allow this freedom the regular expression would become quite elaborate, because each permutation has to be allowed explicitly. This is at the cost of the simplicity and elegance of the regular expression. We have included the full regular expression of the patient information registration in Figure 8.

Although we have not formalized the encodings, it seems the expressive power of the process matrix and regular expressions is the same, but the regular expression representation may be exponential in size compared to the process matrix. There are still advantages to using regular expressions. The first advantage can be seen

directly from the used example. When the same action has to be performed twice, it is necessary to define the same action twice in the process matrix. For example the action *"Check order and preparation match"* is defined twice because both of the nurses has to perform this check. This is not necessary with regular expressions, where we can simply write `N1cmN2cm+N2cmN1cm` to represent that both nurses should perform the check (in any order). Regular expressions are very simple, they have a very intuitive semantics and are already used in many areas such as biology and string processing. Therefore it is very likely that doctors can learn how to express the desired workflows using regular expressions, and many programmers already understand them and thus if regular expressions allow formal verification of implementations, it would make a good candidate as a CPG workflow DSL. Finally regular expressions have been thoroughly studied, which means they have a formal semantics and allows formal reasoning using the semantics, rewritings and finite automaton theory.

Treatment verification

Once a workflow has been expressed as a regular expression, it allows the same reasoning as any other regular expression. In particular it means that a sequence of events (a treatment) can be matched against the regular expression (the workflow). This allows us to automatically check if a treatment complies with a workflow, because we can match the performed sequence of events with the regular expression representing the workflow, and if it matches, then the treatment has been performed as the workflow specifies, and if it does not match then the treatment breaks the workflow.

It is very useful to get a descriptive error message when the verification fails, because this enables us to determine which action broke the workflow and thus who is to blame. If the treatment is matched, it is also useful to get a proper description of how the treatment was matched. This is for example useful when *debugging* the workflow. In case a matched treatment was intended to break the workflow, the matching information is important because it allows us to understand how the treatment was matched and thus if it is an error in the workflow and if so, how it can be corrected. This is where the commonly used tools for regular expression matching are a bit lacking, because most matching algorithms return incomplete information about how the data was matched.

The work we have done on this problem can be separated into three parts. In the first part of Chapter 2 we define a representation called bit-codes of the parse-trees we wish to find. The bit-codes are compact while allowing efficient manipulation of the parse trees. In the first part of Chapter 3 we have investigated, refined, implemented, benchmarked and compared the existing matching algorithms that do return the full matching information, in order to find the most viable method. Many of the performed optimizations are the result of using the bit-code representation in favour of the different original representations which results in a more compact result, noticeably simplified algorithms which use less memory and are more

efficient. In the second part of Chapter 3 we have focused on the most efficient parsing method found in the first part, and optimizes it by adapting Mohris [87] minimization algorithm for sequential transducers, such that it can be applied to the generated output-deterministic transducers.

Specification verification

CPGs are sometimes refined to match the facilities at a specific hospital. If a hospital for example cannot afford to have the doctor register the patient information, the workflow could be restricted to `N1biN1lrN1ph(DcdDsoCPro)*...` where only the nurse is allowed to register the patient information. However, the workflow used at the hospital should comply with the general CPG, and this means in practice that any treatment that complies with the local workflow, should also comply with the global workflow. This is what is usually called language inclusion for regular expressions. In Chapter 2 we develop a new axiomatization of language inclusion for regular expressions. There are many benefits to the new axiomatization, as it enables a deeper understanding of the existing axiomatizations while inheriting some of their properties such as proof-searching methods via encodings (an earlier version [55] includes a coercion synthesis method inspired by the proof search in Grabmayers axiomatization [52]).

Summary

We have now illustrated that regular expressions can be used as a language to communicate workflows between doctors and programmers and that they allow the reasoning necessary to validate treatments and specifications. Some may argue that the representation of workflow events as the conjunction of the participant and action is unnatural because it is possible to write regular expressions that does not represent any workflow, but it is simple and allows for example for easy specification when multiple participants can perform the same action as seen in the example.

The properties that we have not investigated for regular expressions is the ability to formally verify that UI implementations comply with a workflow represented by a regular expression. This would be a sizeable task, because there is no existing way to represent UI implementations in relation to regular expressions. Therefore regular expressions require a lot of basic work before UI verification can even be considered. Instead we have investigated session types, because they already support reasoning that resembles UI verification.

1.4.2 *Session types*

Session types are not as well known, intuitive or elegant as regular expressions, but processes (in the asynchronous π-calculus) can be formally verified to comply with session types via type checking, and therefore it is interesting to investigate if there is a way to represent CPG workflows as session types and UIs as processes, such that the session type compliance exactly corresponds to CPG compliance. Since session types describe the messages that are passed between processes it

should also be possible to describe the information that is exchanged at different points of the CPG workflows, and thus ensure that the necessary information is communicated.

Multiparty asynchronous session types

We have chosen the multiparty asynchronous session types [61] to model workflows, because we wish to represent CPG workflows with more than two participants. Binary session types are substantially more simple than multiparty asynchronous session types, and it is possible to express some sessions with more than two participants by combining binary session types, but the properties guaranteed by the verification are stronger when only one session is used, and the example CPG workflows we are representing does use the additional expressiveness offered by multiparty asynchronous session types.

No representation of CPGs

Session types are designed to represent network protocols, and type-checking verifies that the communication performed by processes does not break the specified protocol. Expressing CPG workflows as session types, and UIs as processes can thus enable a formal verification for UI compliance with CPGs by using the existing type-checking. This turned out not to be as straight forward as hoped. The existing processes had no constructs for expressing UIs, but the real problem was how to express workflows as session types. Since network protocols are described by the allowed sequences of messages and choices, it seems like a good way to represent workflows, but even with the expressiveness of multiparty asynchronous session types, it turns out that many CPG workflows cannot be represented. We found the reason for this to be that some types of interactions in workflows does not occur in network protocols, and therefore the session types cannot express interactions of these types. One of the elements that are missing from session types (and the typed processes) can intuitively be described as *initiative*. Initiative is a vague description, so let us consider a specific simple workflow to better materialize the problem.

Example 7. *Consider a workflow with two participants* `1` *and* `2`*, where each participant has to perform one action* `A` *and* `B` *respectively. If we wish to describe this workflow as a multiparty asynchronous session type, we have to represent each action as a type atom, specificly* `A` *can be represented by* `1` *sending a string to* `2` (`1=>2:1<String>`)*, while* `B` *can be represented by* `2` *sending a string to* `1` (`2=>1:1<String>`)*. Since both of these actions has to be performed, do we represent the workflow as* `1=>2:1<String>;2=>1:1<String>;Gend` *or* `2=>1:1<String>;1=>2:1<String>;Gend`*? They are essentially different, because the first expresses* `A` *before* `B` *and the other expresses* `B` *before* `A`*. We can express that one of the types is selected as a branching type, but which of the types below do we use?*

```
1=>2:1
{^Afirst: 1=>2:1<String>;2=>1:1<String>;Gend,
 ^Bfirst: 2=>1:1<String>;1=>2:1<String>;Gend
}
2=>1:1
{^Afirst: 1=>2:1<String>;2=>1:1<String>;Gend,
 ^Bfirst: 2=>1:1<String>;1=>2:1<String>;Gend
}
```

The difference is whether `1` or `2` decides if `A` or `B` is performed first, but neither case represents the desired workflow, as it is not decided by just one of them. This is what we intuitively referred to as initiative, because we need to express that both `1` and `2` are able to make the first step and perform their action.

Extension

Since most CPG workflows cannot be represented by multiparty asynchronous session types, we have extended the session types and the processes with one construct each. The operation we have added is an abstraction of the procedure to reach a decision by common agreement. In Chapter 4 we formally define the multiparty asynchronous session types with the new type construct, the π-calculus with multiparty sessions with the new process construct and study how to use the session types to model CPGs, how to use the processes to implement workflows, how to verify workflow compliance by type-checking and how the extension affects the expressiveness of the typed process language.

Representing CPGs

The new type construct is called a *symmetric sum type* and is written as a set of branches (using curly brackets around comma separated elements). Each branch consists of an identifier (called a label) followed by a colon and a type expressing how the session proceeds if this choice is used. Unlike the existing session type constructs, there is no sender and receiver and therefore it is *symmetric* because it *looks the same* to each participant. To represent the workflow from Example 7 we write:

```
{^A: 1=>2:1<String>; {^B: 2=>1:1<String>;Gend},
 ^B: 2=>1:1<String>; {^A: 1=>2:1<String>;Gend}
}
```

The workflow is represented by a choice. If `A` is selected, then the action `A` is performed before a new choice where `B` must be selected and performed. If `B` is selected, then the action `B` is performed before a new choice where `A` must be selected and performed. In this way both `A` and `B` are performed, they can be performed in any order, and the decision is not made by a single participant but by common agreement. Common agreement is not exactly the same as initiative. If a participant wishes to perform an action the difference is whether the remaining participants have to actively accept this action, or actively reject the action. Since the difference is in the default choice of acceptance for the passive participants this difference can be compensated by the UI implementation.

The new process construct is called *symmetric synchronization* and is written as the keyword sync with a subscript containing an integer (the number of participants in the synchronization) and a session, followed by a set of labelled processes (curly brackets containing comma separated elements), where a labeled process is written as a label followed by a colon and a process expressing what to do if this choice is used. Processes implementing the example workflow are in Figure 9a.

UI Specification

Figure 9a specifies the process for each participant, but they do not specify the UIs for these decisions. Thus the choices are not be made by a user but by the synchronization implementation. This is reflected in the process semantics where the decisions are made *randomly*. Because we are interested in specifying and verifying UIs we have extended the described synchronization with UI specifications. The processes already specifies the choices in each decision, so the information needed to specify the UIs is the input fields for each choice for each participant. This extension results in a new synchronization process construct called `guisync` and is a very simple extension to the `sync` construct, which allows the process to specify input fields for each choice. The `guisync` construct is very similar to the `sync` construct, but takes an extra argument `p` which is the participant number of the session (used to identify the UI), and each branch has a (possibly empty) list of simply typed arguments which are used as the UI input fields. The stepping and typing rules of `guisync` are also very similar to the rules for `sync`, except that the given participant number is checked, and the input values are type-checked. Also the stepping rule has a sidecondition (†) which should state that the user for each participant `p` has accepted the chosen branch (`h`) and given the used input $\tilde{v}_{\mathtt{hp}}$. The rules for `guisync` are given in Appendix A.2.1. Finaly we have added a simple process constructor `guivalue` which can be used to add information to the UIs.

Using the described `guisync` and `guivalue` constructs, the user interfaces for the simple workflow can be specified as in Figure 9b, and the initial user interfaces for both participants are illustrated in Figure 9c.

In order to illustrate the representation of workflows and UIs for real CPGs, we include the representation of the workflow from Example 6 and the UI for each participant.

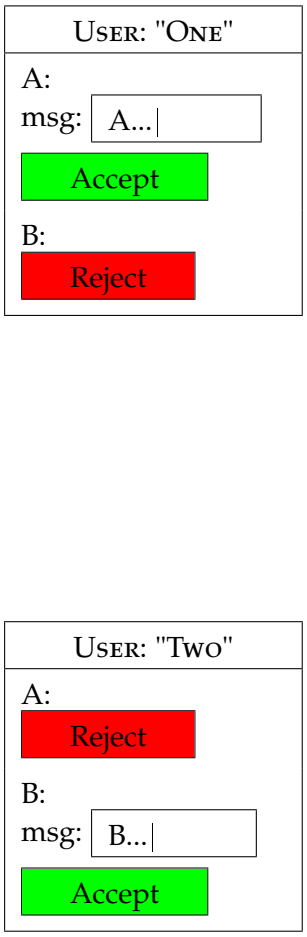
Example 8. *In Chapter 4 we describe how the session type and processes can be automatically generated from the process matrix used to specify the workflow. The result would consist of a single session and thus the type checking would be able to guarantee deadlock freedom. The result would however be too large to include.*

With the given workflow, it is possible to design the session type in a more modular way, where the workflow is divided into sub-workflows for each ward, which are combined by the porters and receptions using delegation. This design is more intuitive, because the workflow is divided into the groups that actually work together, and the used porters are modelled directly. This representation is the result of manually reading the workflow description,

Figure 9 The relation between synchronization and UI specification

```
// User One
sync(2,s)
{^A:
  s[1]<<"A...";
  sync(2,s)
  {^B:
    s[1]>>x;

    end
  },
 ^B:
  s[1]>>x;

  sync(2,s)
  {^A:
    s[1]<<"A...";
    end
  }
}
| // User two
  sync(2,s)
  {^A:
    s[1]>>y;

    sync(2,s)
    {^B:
      s[1]<<"B...";
      end
    },
   ^B:
    s[1]<<"B...";
    sync(2,s)
    {^A:
      s[1]>>y;

      end
    }
  }
```

```
guivalue(2,s,1,"User","One");
guisync(2,s,1)
{^A(msg:String="A..."):
  s[1]<<msg;
  guisync(2,s,1)
  {^B():
    s[1]>>x;
    guivalue(2,s,1,"x",x);
    end
  },
 ^B():
  s[1]>>x;
  guivalue(2,s,1,"x",x);
  guisync(2,s,1)
  {^A(msg:String="A..."):
    s[1]<<msg;
    end
  }
}
| guivalue(2,s,1,"User","Two");
  guisync(2,s,2)
  {^A():
    s[1]>>y;
    guivalue(2,s,1,"y",y);
    guisync(2,s,2)
    {^B(msg:String="B..."):
      s[1]<<msg;
      end
    },
   ^B(msg:String="B..."):
    s[1]<<msg;
    guisync(2,s,2)
    {^A():
      s[1]>>y;
      guivalue(2,s,1,"y",y);
      end
    }
  }
```



a) Synchronization **b) UI Specification** **c) Initial UIs**

and expressing the workflow aspects of it as a multiparty asynchronous session type. It is therefore not formally a semantically equivalent encoding.

The implementation containing the session types and UI implementations is in Appendix A.2.2.

Used paradigms

The implemented workflow uses a number of objects such as flowcharts (not to be confused with the flowchart notation for workflows), workslips and drugs, that can be created and should be handled in a certain way, where some of the restrictions are specified directly in the workflow. For example a drug should first be prepared, then checked a certain number of times before being admitted to a patient. In an object oriented language, these objects would be specified by classes, allowing the objects to be created with the `new` command, but some restrictions such as requiring a drug object to be checked a certain number of times before being admitted to a patient are very hard to represent in this way. In our framework, we have to specified the objects with session types and implement a service which basically corresponds to the class implementation in an object oriented language. This allows new objects to be created as sessions by connecting to the implemented service. In Example 8 the drug service is implemented on line 1 to 126, the workslip service is implemented on line 668 to 743, while the flowchart service is implemented on line 128 to 436.

The workflow can be specified by one large session type, but this has some disadvantages. If for example the workflow for the pharmacy is changed, all participants have to update to the new session type and re-verify that the workflow is respected. In this way it becomes a global problem to make a local changes. It is also very intuitive to describe the workflow for the oncology ward and the pharmacy separately, as they are in different places and involve different people. Finally it may be more efficient to describe the workflows separately, both in the size of the type because the combined workflow may have a lot more non-determinism, in the size om the implementation because the actions in the other workflows does not have to be handled, and in the execution because each synchronization may require less communication. In Example 8 the specification of the oncology ward workflow is on line 174 to 199 and 510 to 524, while the specification of the pharmacy workflow is on line 745 to 851.

In order to describe how the separately described workflows interact, we simply simulate the interaction that occurs in real life. For example when the pharmacy has prepared and verified a drug, the drug is transported to the oncology ward by a porter. Therefore we implement a porter service, such that the people at the pharmacy can connect to this service to get a porter session (they look for an available porter), they can then give the drug to the porter using delegation, and the porter will connect to the oncology ward to deliver the drug. In Example 8 the porter transporting the drug is implemented on line 493 to 507. This also shows the power provided by delegation, as it can be used to describe the exchange of objects and responsibilities.

Assertions

We have now represented a real world workflow as a session type, and expressed the UIs of each participant in a way such that it can be automatically verified to

comply with the workflow in Example 8.

We believe however, that there are many workflows where it is not practically feasible to use the described representation. The reason why this workflow can be represented is that it is relatively *deterministic*, meaning that at each point, the workflow has relatively few choices. Just like we observed with regular expressions, each allowed path through a workflow has to be allowed explicitly, and therefore workflows with a high level of non-determinism can become very large.

The session types allow the workflow specification to include what information is exchanged at different points in the workflow, but only the types of the message can be specified. In some cases it would be very useful to specify more than just the type. For example when a doctor prescribes some medication to a patient, it would be very useful to specify that the prescribed dose should be less than the lethal amount. A simpler constraint could be when the same information is given to two participants, we would like to specify that the sent messages are actually identical. In Chapter 5 we investigate the existing work on assertions in the context of session types [20] and merge this work with our own symmetric synchronization.

Apims implementation

The asynchronous pi-calculus with multiparty sessions and symmetric synchronization (aπms) and its typesystem forms a theoretic foundation that can be used to express UI implementations, and verify by type-checking that they comply with the workflows in CPGs. This is however purely theoretical, and the expressed UIs cannot be executed without transferring them from the mathematical model (aπms) to an executable language. The more the destination language differs from aπms, the more complicated it is to define such a translation and ensure that the semantics is preserved. Therefore we have defined Apims, an `ascii` language which is closely related to aπms.

We have implemented a parser, a type checker and an interpretor for Apims [1]. This enables automatic verification and execution of the UIs defined in aπms.

1.5 FIELD STUDY

Objectives

In the first week of November 2010, the yellow team at the section of urology at Rigshospitalet was kind enough to allow us to observe their work. This has served many purposes in the presented project. First of all we believe it is important to get acquainted with the domain we are designing models and frameworks for. This field study has allowed us make simple observations like what tasks are performed, the way work is coordinated, what and how information is exchanged and finally what tools (including software) that are currently used and how. Finally the field study has provided us with a real world workflow, we can model in the developed framework to test its capabilities.

THE WORKFLOW We start by giving a rough description of the observed workflow, as this provides a context for the other observations. The ward has a number of rooms with two beds in each, a reception and a shared office with desktop computers. There are also different supply rooms, and in particular a medication storage with a designated workspace for preparing the medication, with a computer connected to the medication prescription and administration system. Each day is divided into three work-shifts from 7am to 3.15pm, from 3pm to 11.15pm and from 11pm to 7.15am. The shifts overlap by 15 minutes, to allow the nurses from the ending shift to give the nurses from the starting shift an update on each patient, special overview tables with one row for each patient are used for this. We observed the shift from 7am to 3.15pm, because we believe this is where the most interesting parts of the workflow occur.

The overall structure of the shift is as follows

7.00 Work shift

- Delegation of patients
- Nurse rounds (breakfast is served and registered)
- Morning medicine administration

8.30 Doctors arrive

- Doctors are updated
- Doctors rounds (todays actions are planned)
- Journals are updated

10.00 Doctors leave

- Nurse rounds (todays actions are performed)

14.00 Afternoon medicine administration

- Nurse rounds (Prepare overview for next shift)

15.00 Work shift

OBSERVATIONS Our observations are too numerous to include them all. This is unfortunate, especially because the small details are some of the most valuable observations, as they are usually omitted from the explanations of workflows, and it is hard to know what details to ask about in interviews. We can only recommended that such a field study is considered by all who plans to develop such frameworks. We have selected some of the key observations that we made during the field

Figure 10 Example of delegation planning notation

Room	Patient	Nurse	Comments
1a	p1	n1	
1b	p2		operation at 2pm
2a	p3		
2b	p4		discression ..., discharged
3a	p5	n3	
3b	p6		
4a	p7		
4b	p8	n2	
	p9		operation at 10am - on the way back
	p10		discharged
	p11	n3	arrival 3pm
	p12	n3	arrival 3pm

study, which we will discuss as they give an extra insight into the challenges, and applications for the frameworks. The observations that are not included have still played a vital role in designing the created frameworks, discovering and prioritizing future work.

Delegation of patients

In the office of the ward, the nurses have a whiteboard with a permanent table. The table has two rows for each room on the ward (one for each bed), and some extra rows for incoming patients and patients in the hall (in case there are not enough rooms). The table has columns that allows the patient name, the attending nurse and comments to be assigned to each row. An example of such a table is given in Figure 10. The comments contain the plans, appointments and other important information, some examples are given in the table. In the beginning of every shift, the nurses divide the patients between them, and writes the agreed division in the **Nurse** column. This deserves special attention, as this is not a simple assignment of nurses. The reason for this is that some shifts are overlapping, and some nurses have to take breaks for education and other duties. Therefore the nurses have to plan who takes over in the periods where a nurse is absent. In the given table, it could be the situation that the shift starts at 7am, but nurse n3 arrives later (for example 10am). The other nurses (n1 and n2) have divided the patients between them (except the new patients that are due to arrive after nurse n3), and when nurse n3 arrives she will take over the patients p5, p6 and p7. We can consider how to model the nurse assignment. If the treatmen of each patient is represented as a session, then the nurse assignment describes the delegations that should be carried

out. When the shift starts, the patient sessions are delegated from the nurses of the previous shift to the assigned nurses in the new shift, and when nurse n3 arrives the patients p5 p6 and p7 are delegated from nurse n1 and n2 respectively.

Preparation of medicine

The medicine preparation is a highly sensitive step, where small mistakes can have grave consequences. Therefore many precautions have been made in this process. As mentioned there is a specific workplace for the preparation, where all the necessary drugs, manuals and tools are collected. Before a nurse goes to prepare the medicine, she ensures that she has no tasks in the immediate future, and asks a coworker to cover her patients in case something should happen. These precautions are to ensure that there will be no interruptions in the preparation, as studies have shown one of the main causes for adverse events to be interruptions [3]. The effort to avoid interruptions is observed for other tasks as well, where the nurse starts by collecting all the necessary items on a cart, so they can be brought to where the task is performed. A software system can help by alerting the nurse if she selects to prepare the medicine while she has pending actions in the immediate future, and the system could list the items necessary to perform each task, so they can be collected before initiating the task. In this way the software can help avoid the interruptions that are suspected of causing many adverse events.

Updating journals

When the doctor has visited a patient in the doctor rounds, he updates the journal. To save time he does not write the changes manually, in stead he dictates the changes on a software system. After the doctor rounds, the journals are then carried to the medical secretaries who updates the journals based on the dictations they can find in the dictation system. This means that the updates are sent using a software system, while the journals to be updated are physically carried to the secretaries and back. This is one place where the digitalization of journals can potentially increase the efficiency, as there would be no need to carry the journals forth and back.

Task automation

Our observations revealed a number of tasks that could be automated if the workflow was digitalized. For example the fluid and energy balance of the patients are monitored, which means the amount of fluid the patients drink (and in some cases the amount of fluid excreted) is logged in a form. The form has room for a week, so each week the data is transfered to another form, by summarizing the numbers on the form. This is done manually often using a calculator, and this step could easily be automated if the data was entered directly in a it-system. After the data is transferred, the original form is destroyed to avoid it from being included twice.

Discharging requirements

An interesting restriction when modelling the workflow is that patients should not be discharged until they have sucessfully gone to the toilet. This is of course because the patients are in this ward because they have problems in this region and it is possible – especially if people have undergone surgery – that the passage can be blocked. This is easily ruled out if the patient can successfully go to the toilet,

and therefore this is a requirement before discharging the patient to ensure that such problems are discovered.

Specialized systems

The last observation we will mention is on the software that is currently used. The ward uses a collection of specialized software systems, each performing a very specific task. Examples of these systems are the medicine prescription and administration system, the operation room scheduling system, the blood test ordering system and the dictation system. This is probably a good idea for developing the software, as each project has a limited size, but it is a problem to use when the systems do not interact well. For example the nurses and doctors have to log out of the old system and log on to the new system every time they change task. This was obviously frustrating for the staff, and it seemed there were many planned projects on better integration between the different systems.

MODEL AND IMPLEMENTATION We have used the observed workflow to test the modelling capabilities of Apims, and the result is in Appendix A.2.3. We have used the extension with assertions from Chapter 5 extensively in this representation, as it would have been infeasible to describe the types (and processes) without assertions, there is simply too much non-determinism in the workflow to describe each allowed path explicitly. We have used the same paradigms as mentioned in Section 1.4.2, for example the patient journal is implemented as a session that can be created by connecting to a journal service. The doctors and nurses have been implemented to wait for a connection where they receive the patients journal, after which they will execute the workflow. In this setting the receptionist of the ward can be implemented by waiting for a patient, when a patient arrives, a journal can be created using the information provided by the patient, then the receptionist connects to the doctor and the nurse to create a treatment, sends the journal to the nurse, and delegates the obtained treatment to the patient. This illustrates the use of delegation, and in this way it can provide the receiver with a guarantee, in this case the patient is guaranteed a treatment when the treatment session is delegated to him. The implementation of the receptionist is on line 763 to 783 and is included in Figure 11, and a screenshot of the receptionist UI for an example patient is in Figure 12. A graphical illustration of the enrolment process is on the titlepage of Chapter 5.

1.6 RELATED WORK

REGULAR EXPRESSIONS Regular expressions may seem very limited as a workflow language, but the representation of workflows we have illustrated extends to more expressive frameworks like context free grammars [74]. Also many frameworks (such as regular expression types [64] and session types [61] are essentially extensions of regular expressions, and therefore the development of regular ex-

Figure 11 Implementation of receptionist

```
def Receptionist=                          // Receptionist service
                                           // implementation
  link(2,sReception,s,2);                  // Wait for new patients
  s[2]»name;                               // Receive data from paitent
  s[2]»cpr;
  s[2]»symptoms;
  guivalue(2,s,2,"Name:",name);            // Display data in UI
  guivalue(2,s,2,"CPR:",cpr);
  guivalue(2,s,2,"Symptoms:",symptoms);
  guisync(2,s,2)
  { Enroll(room="Room12a": String):        // Enroll the patient
    link(2,journal,j,1);                   // Create journal
    j[1]«name;
    j[1]«cpr;
    j[1]«room;
    j[1]«symptoms;
    link(3,treatment,b,1);                 // Find doctor and nurse
    b[3]«j;                                // Give journal to nurse
    s[1]«b;                                // Delegate treatment
    Receptionist()                         // Wait for new patients
  }
in Receptionist()
```

pression reasoning is the first step in developing the same reasoning for such frameworks.

There is more related work for regular expressions in Section 2.5.

MULTIPARTY ASYNCHRONOUS SESSION TYPES The type verification we offer does not guarantee progress for multi session processes, because deadlocks are possible. Progress is very desirable, and there are two lines of work that can help guarantee deadlock freedom. The first is a more restrictive typing judgement for multiparty asynchronous session types [17], that can guarantee progress, and the other is a process analysis to verify deadlock freedom [75], although this currently only works for binary sessions.

π-calculus implementations

There are existing implementations of the asynchronous π-calculus and session types. Prominent examples are Pict [100], Mobility workbench [120], SJ [66] and Occam-pi [123]. We are however not aware of any implementations which allow multiparty sessions or verification of multiparty asynchronous session types.

join-calculus

Figure 12 Screenshot of receptionist GUI

There are many variants of the π-calculus. One that is particularly interesting in our optic is the join-calculus. It has been designed to be efficient while preserving the expressiveness of the asynchronous π-calculus, but the interesting part is the way it extends the message reception which is of particular relevance to our objective. The extension allows a process to wait for messages on a set of channels in stead of a single channel. If we recall Example 7 the problem in expressing the UIs was exactly that the UI process needs to be able to express initiative, which may be related to being able to listen to multiple channels simultaneously. Therefore it would be interesting to investigate if the UIs can actually be represented in the join-calculus, although it is unlikely that they can be represented as compact and elegantly as they can using the symmetric synchronization extension. As there is no version of multiparty session types for the join-calculus, there is no obvious way to represent the workflows in the same way as we do using multiparty session types with symmetric sum types. The only type-system for the join-calculus that we are aware of is an ML-like type-system which has no notion of sessions [43]. As the join-calculus has been designed for efficient implementation, it is not surprising that there are a lot of implementations out there, including JoCaml [31], Parallel C# [4] and Boost.Join [2].

Multicore utilization

The reason why we have chosen the asynchronous π-calculus to express the UI processes is that it allows each participant to be described as an individual process, and allows all the processes to work together to obtain a common goal which is to perform the treatment of a patient. This property can be used for other purposes than to verify workflow coherence, and the most popular application is to write programs that efficiently utilizes multiple cores, processors or servers. This is becoming an increasingly popular and even inevitable programming feature and many frameworks exists to allow this efficiency. The most popular is probably MPI

[54] which again has many implementations where OpenMP [19] is believed to be the most efficient and thus the most used.

There is more related work for session types in Section 4.6 and 5.5.

WORKFLOW MODELS There are many existing workflow models, some of which are introduced in Section 1.3. There are similarities between the existing models and the presented session type model, but all the investigated models are built on the CIG principle, which means that the guideline can be interpreted directly, and thus the UI design and implementation has to be injected into the model somehow. We have separated the UI specifics from the workflow model by using the model to verify an implementation as opposed to executing the model directly. None of the investigated models allow static verification of processes or UI implementations via type-checking. In stead they either offer no safety, or they interpret the model directly which of course ensures that the implementation complies with the model as the implementation and the model is the same thing.

We believe that the separation of the workflow model in the types from the UI implementation and design in processes allows a greater flexibility and expressiveness in the UIs while the workflow design remains uncluttered by the UI specifics, and the static verification provides almost the same safety that is guaranteed by direct model interpretation.

YAWL

Yet another workflow language (YAWL) [116] is a language aimed at business process implementation. In many ways YAWL is comparable to BPEL, but where BPEL is developed by an industry standard committee, YAWL is the academic counterpart. The key difference in functionality is that YAWL has a formal semantics, and this enables some static analysis. Woflan [119] is a tool to statically verify workflows modelled in YAWL. The analysis is based on the petri net semantics of YAWL. The errors found are not model compliance errors but possible deadlocks and similar errors, however the analysis is not complete in the way that some errors are not detected.

1.7 FUTURE WORK

Regular expressions as types

REGULAR EXPRESSIONS A promising application of our work on regular expressions is to use regular expressions as datatypes. The types denoted by regular expressions can be considered as string variants. The use of bit-codes can ensure a compact representation of the program data and thus a small memory footprint, while data can still be processed efficiently and for example allow efficient pattern matching. The structure of regular expressions can allow programs to express string processing in an intuitive way like in Perl [121], and by using the developed coercions, data can be translated between types efficiently, such that the type that allows the most intuitive description of the program can be used.

COMBINING REGULAR EXPRESSION AND SESSION TYPE THEORY In the presented work, the results for regular expressions may seem completely unrelated to the work we have done on session types, but they are related in many ways and some of them could be explored in future work.

Uniting the typesystems

One application of our work on regular expressions in session types is simply to allow regular expressions as types in the simple types S used in the multiparty asynchronous session types. Besides the possibilities mentioned above, these types allows the types expressed to be more specific, so they can express the structure of the strings communicated. The compact bit-codings can also be used when sending and receiving messages, so the bandwidth used could also be reduced by this union of the type-systems.

Subtyping for session types

There is currently no proper notion of subtyping for multiparty asynchronous session types. We have allowed some flexibility by allowing optional labels in the symmetric sum types, and the original types uses a very basic notion of subtyping by label set inclusion. Some papers define subtyping for binary session types [48], other variants of multiparty session types [98] or other kinds of subtyping for multiparty asynchronous session types [88], but to our knowledge there is no existing definition or subtyping algorithm for the multiparty asynchronous session types we have extended. It would therefore be very useful to give a proper notion of semantic subtyping, and an algorithm to decide subtyping. Such work could be based on the method in Chapter 2 where we do the same for regular expressions. In case the subtyping results in changes in the messages the coercions that are provided by our method could result in translating agents working as a proxy translating messages between the more general and the more specific protocol.

If we want to do a quick and dirty subtyping, we can convert the session types to regular expressions using an erasure like the one defined below.

$$\begin{aligned}
\mathrm{RE}[\![\mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{k}\langle \mathsf{M}\rangle; \mathsf{G}]\!] &= \mathrm{RE}[\![\mathsf{G}]\!] \\
\mathrm{RE}[\![\mathsf{p}_1 \to \mathsf{p}_2 : \mathsf{k}\{\mathsf{l} : \mathsf{G}_\mathsf{l}\}_{\mathsf{l}\in\mathsf{L}}]\!] &= \Sigma_{\mathsf{l}\in\mathsf{L}}\, \mathsf{l}\mathrm{RE}[\![\mathsf{G}_\mathsf{l}]\!] \\
\mathrm{RE}[\![\{\mathsf{l} : \mathsf{G}_\mathsf{l}\}_{\mathsf{l}\in\mathsf{M};\mathsf{O}}]\!] &= \Sigma_{\mathsf{l}\in\mathsf{M}\cup\mathsf{O}}\, \mathsf{l}\mathrm{RE}[\![\mathsf{G}_\mathsf{l}]\!] \\
\mathrm{RE}[\![\mu\mathsf{X}.\mathsf{G}]\!] &= \mu\mathsf{X}.\mathrm{RE}[\![\mathsf{G}]\!] \\
\mathrm{RE}[\![\mathsf{X}]\!] &= \mathsf{X} \\
\mathrm{RE}[\![\mathsf{end}]\!] &= 1
\end{aligned}$$

This erasure produces a tail-recursive μ-term which can then be converted to an equivalent regular expression. This allows us to use a decision algorithm for language inclusion of regular expressions to find if the results are related. The result will not correspond to semantic subtyping of session types because parts of the session types are lost in the erasure, but the procedure will decide if the possible sequences of choices in one session type are also possible in the other session type. Moreover this process can be used to relate regular expressions to

session types. This is particularly useful because session types are too technical to expect doctors and domain experts to write the desired CPGs as session types, but regular expressions are simple enough to allow this. Therefore if the CPG is expressed as a regular expression, but the implementation is verified to comply with a session type, then we are interested in verifying that compliance with the session type means that only the sequences of choices allowed by the RE describing the CPG are possible, and the suggested erasure combined with regular expression language inclusion might allow this verification.

APIMS DEVELOMPENT Our experience with modelling and implementing the example workflows (and many other programs) in Apims is that many errors are found by the type-checker. Apims is a prototype language, and thus it is not very intuitive or easy on the eye. This means that many programming errors were made during the implementation which meant it took some time to get the implemented processes to type-check. Once an implementation had been verified it actually worked, and we have executed many simulations on the implemented workflows in search of errors that were undetected by the type-checker, but we did not find any. There are however some pitfalls that Apims programmers should be cautious to avoid, and these pitfalls could be targeted by future work. First of all the global types used have to be linear and coherent. The coherence is checked by Apims when a global type is projected, but linearity is currently not checked. There is no profound problems in checking linearity, but until it is implemented programmers should ensure manually that the used global types are linear. Even though type-checking verifies that the messages sent are of the correct type, there is no guarantee that the message contains the correct value. Using assertions this can be ensured for boolean messages, but otherwise one has to be careful, especially not to swap two messages of the same type. Consider for example the type $1\rightarrow 2{:}1\langle\text{String}\rangle;1\rightarrow 3{:}2\langle\text{String}\rangle;\mathsf{end}$. In this case participant one should be careful not to send the message for participant 2 to participant number 3 and vice versa. As the type-checking does not ensure progress for multi-session processes, the programmer has to avoid deadlocks manually. One error that is easy to make is to declare a service, but forgetting to implement it. In this case any process trying to use the service will be stuck waiting for the service to accept its connection. Finally the type-checking relies on a cooperative setting. This means for example that a process is not forced to send the message it is expected to by the session type. The type-checking guarantees that the process cannot terminate before it has sent its pending messages, but it can just go into an infinite loop. It is however unlikely that a programmer would implement this behaviour by accident, and thus these errors only occur in an adversarial setting.

UI Usability

The UI implementations are currently very simple. They can display information and allow a set of actions with typed input fields. It should be possible to extend

the UI functionality to allow specification of the visualization without affecting the UI verification. This could increase the usability by emphasizing important actions, or build a menu system for navigating in case of many possible actions. One way to do this could be by applying activity based computing (ABC) [16]. The current UI module receives the information about sessions, participants and their choices, so the users can see the session they participate in, and for each session get a list of possible actions to execute. The basic idea of ABC would be that the users in stead of a list of sessions would get a list of activities, and for each activity they would get a list of actions (across all sessions) that can be performed in that activity. There could for example be an activity called medicine administration and one called doctor rounds. This way the usual activities can be performed without having to use a lot of time to switch between the different session UIs and find the desired action in a long list of possible actions. This could result in a UI that is easier and faster to use, which is vital for the frameworks to be used without wasting the time of the clinical staff.

Deadlines

WORKFLOW EXTENSIONS The field study revealed that many actions have real time constraints. The medication administration, doctors rounds and other actions are performed at specific times of the day, and it would be advantageous to be able to model this with deadlines and other real-time constraints. There is a problem enforcing deadlines in UI implementations, as there can be no guarantee the user will act within a given deadline. This means deadlines for UIs cannot be statically verified, but by including deadlines and other real-time constraints in the session type and processes, these constraints could at least be visualized in the UIs.

1.8 CONCLUSIONS

We will now draw some conclusions, based on the results of the project. We start by considering the theses from the abstract.

Thesis 1

In Chapter 3 we have modified multiple algorithms for regular expression parsing, to produce compact bit-codes without explicitly materializing the parse-trees. As observed in Chapter 3 the original algorithms were asymptotically efficient, and the modifications simplify and optimize the algorithms. Therefore Thesis 1 has been substantiated.

Thesis 2

In Chapter 4 we have extended the multiparty asynchronous session types with symmetric sum types, which represents a specific type of social interaction, and showed by example that some workflows can be represented by the extended type-system, and that processes can be verified to comply with the workflow by type checking. Therefore Thesis 2 has been substantiated.

Thesis 3

In Section 1.4.2 and 1.5 we have showed by example, that workflows in general and in particular some real world CPGs can be represented as session types

extended with symmetric sum types and assertions, and that UI implementations can be verified to comply with the represented workflows via type checking. Therefore Thesis 3 has been substantiated.

Regular expressions are simple, elegant and intuitive, and this make them ideal for communicating with non-computer scientists. We have demonstrated that regular expressions can be used to express workflows. The reasoning regular expressions provide for CPG workflows is treatment verification via matching, and specification verification via language inclusion. There is however currently no way to verify UI compliance for regular expressions.

Multiparty asynchronous session types however, are rather technical and unintuitive, but we have showed that using the extensions with symmetric sum types and assertions, they can represent CPG workflows, and UI implementations as well as automated processes can be verified to comply with the represented workflows via type checking.

In Section 1.7 we have suggested an investigation to combine regular expressions and session types to achieve the best of both worlds.

In this project, we have presented many theoretic results and applied them to verification of CPG workflow compliance. We have also implemented the developed algorithms and languages, which has enabled us to test the developed theory.

The list of academic contribution of this project is in Section 1.2.

Part II

REGULAR EXPRESSIONS

2

REGULAR EXPRESSION CONTAINMENT: COINDUCTIVE AXIOMATIZATION AND COMPUTATIONAL INTERPRETATION

Authors:

Fritz Henglein - University of Copenhagen,
Lasse Nielsen - University of Copenhagen

Presented at:

POPL 2011 – 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages

We present a new sound and complete axiomatization of regular expression containment. It consists of the conventional axiomatization of concatenation, alternation, empty set and (the singleton set containing) the empty string as an idempotent semiring, the fixed-point rule $\mathsf{E}^* = 1 + \mathsf{E} \times \mathsf{E}^*$ for Kleene-star, and a general coinduction rule as the only additional rule.

Our axiomatization gives rise to a natural computational interpretation of regular expressions as simple types that represent parse trees, and of containment proofs as *coercions*. This gives the axiomatization a Curry-Howard-style constructive interpretation: Containment proofs do not only certify a language-theoretic containment, but, under our computational interpretation, constructively transform a membership proof of a string in one regular expression into a membership proof of the same string in another regular expression.

We show how to encode regular expression equivalence proofs in Salomaa's, Kozen's and Grabmayer's axiomatizations into our containment system, which equips their axiomatizations with a computational interpretation and implies completeness of our axiomatization. To ensure its soundness, we require that the computational interpretation of the coinduction rule be a hereditarily total function. Hereditary totality can be considered the mother of syntactic side conditions: it "explains" their soundness, yet cannot be used as a conventional side condition in its own right since it turns out to be undecidable.

We discuss application of *regular expressions as types* to bit coding of strings and hint at other applications to the wide-spread use of regular expressions for substring matching, where classical automata-theoretic techniques are *a priori* inapplicable.

Neither regular expressions as types nor subtyping interpreted coercively are novel *per se*. Somewhat surprisingly, this seems to be the first investigation of a general proof-theoretic framework for the latter in the context of the former, however.

2.1 INTRODUCTION

What is *regular expression matching*? In classical theoretical computer science it is the problem of *deciding* whether a string belongs to the regular *language* denoted by a regular expression; that is, it is *membership testing*. In this sense, `abdabc` matches `((ab)(c|d)|(abc))*`, but `abdabb` does not. This interpretation is used for most theoretical computer science results: NFA-generation, DFA-generation by subset construction, DFA-minimization, the Myhill-Nerode Theorem, the Pumping Lemma, closure properties, the Star Height Problem, Brzozowski derivatives, fast regular expression equivalence algorithms and matching algorithms, coalgebraic characterizations, bisimulation, etc. If membership testing is all we are interested in, regular expressions and finite automata denoting the same *language* are completely interchangeable. In that case we may as well implement regular expression matching using a state-minimized DFA and forget about the original regular expression.

In *programming*, however, membership testing is rarely good enough: We do not only want a yes/no answer, we also want to obtain proper *matches* of substrings against the constituents of a regular expression so as to *extract* parts of the input for processing. In a *Perl Compatible Regular Expression (PCRE)*[1] matcher, for example, matching `abdabc` against `E = ((ab)(c|d)|(abc))*` yields a substring match for each of the 4 parenthesized subexpressions: They match `abc`, `ab`, `c`, and ϵ (the empty string), respectively. If we use a POSIX matcher [37] instead, we get `abc`, ϵ, ϵ, `abc`, however. How is this possible? The reason is that $((ab)(c|d)|(abc))*$ is *ambiguous*: the string `abc` can match the left or the right alternative of $(ab)(c|d)|(abc)$, and returning substring matches makes this difference observable. In a membership testing setting ambiguity is not observable and thus not much studied.

An oddity and limitation of Perl-style matching is that we only get one match under Kleene star, the last one. This is why we get a match of `abc` above, but not `abd`. Intuitively, we would like to get the *list* of matches under the Kleene star, not just a single one. This is possible, with *regular expression types* [64]: Each group can be named by a variable, and the output may contain multiple bindings to the same variable. For a variable under *two* Kleene stars, however, we cannot discern the bindings between the different level-1 Kleene-star groups. An even more refined notion of matching is thus *regular expression parsing*: Returning a parse tree of the input string under the regular expression read as a *grammar*.

A little noticed fact is that the parse trees for a regular expression are isomorphic to the elements of the regular expression read as a *type*; e.g. the type interpretation $\mathcal{T}[\![E]\!]$ of regular expression $E = ((ab)(c|d)|(abc))*$ is $((a \times b) \times (c + d) + a \times (b \times c))$ list with a, b, c, d being singleton types identified with the respective values

1 See `http://www.pcre.org`.

a, b, c, d they contain. The *values* $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inr}\,(a, (b, c))]$ and $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inl}\,((a, b), \mathrm{inl}\,c)]$ are elements of $((a \times b) \times (c + d) + a \times (b \times c))$ list, representing two different parse trees of the same type. Since their *flattening* (unparsing) yields the same string $abdabc$, this shows that $((ab)(c|d)|(abc))^*$ is grammatically ambiguous.

When we have a parsed representation of a string we sometimes need to transform it into a parsed representation of another regular expression. Consider for example E_d = `(ab(c|d))*`, which is equivalent to E. E_d corresponds to a DFA that can be used to match a string efficiently. But what if we need a parsed representation of the string with respect to E? We need a *coercion*, a function that maps parse trees under one regular expression (here E_d) into parse trees under another regular expression (here E) such that the underlying string is preserved. Since E is ambiguous there are different coercions for doing this. The choice of coercion thus incorporates a particular ambiguity resolution strategy; in particular, we may need to make sure that it always returns the *greedy left-most* parse, as in PCRE matching, or the *longest prefix* parse, as in POSIX matching. Also, we will be interested in favoring *efficient* coercions over less efficient ones amongst extensionally equivalent ones; e.g., for coercing E to E, we prefer the constant-time coercion that copies a reference to its input instead of the linear-time coercion that traverses its input and returns a copy of it. Even if coercions are not used to transform parse trees, they are useful for regular expressions under their language (membership testing) interpretation: The existence of a well-typed coercion from $\mathcal{L}[\![E]\!]$ to $\mathcal{L}[\![F]\!]$ is a proof object that logically certifies that E is contained in F. Once it is constructed, it can be *checked* efficiently for ascertaining that E is contained in F instead of embarking on search of a proof of that each time the containment needs to be checked.[2]

The purpose of this paper is to develop the basic theory of *regular expressions as types* with *coercions* interpreting containment as a conceptual and technical framework for regular expression based programming where the classic language-theoretic view is insufficient.

2.1.1 *Contributions*

Before delving into the details we summarize our contributions.

Regular expressions as types

The interpretation of regular expressions as types built from empty, unit, singleton, sum, product, and list types was introduced by Frisch and Cardelli [46] for the purpose of regular expression matching. We allow ourselves to observe and point

2 The size of a coercion will necessarily be exponential in the sizes of E and F for complexity-theoretic reasons in the worst case, but it *may* be small in many cases.

out that the elements of regular expressions as types correspond exactly to the parse trees of regular expressions understood as grammars. Frisch and Cardelli refer to types as describing "a concrete structured representation of values", but do not verbalize that those representations are essentially parse trees. Conversely, Brabrand and Thomsen [22] as well as other works define an inference system for parse trees, but do not make explicit that that is tantamount to a type-theoretic interpretation of regular expressions.

Proofs of containment by coercion

We observe that containment can be *characterized* by finding a *coercion*, a function mapping every parse tree under one regular expression to a parse tree with same underlying string in the other regular expression.

This means that proving a containment amounts to finding a coercion for the corresponding regular types, allowing us to bring functional programming intuitions to bear. For example, $\mathsf{E} \times \mathsf{E}^* \leqslant \mathsf{E}^* \times \mathsf{E}$ for all E can be proved by defining the obvious function `f`

```
fun f : 'a * 'a list -> 'a list * 'a
```

that retains the elements in the input.

The idea of a coercion interpretation of an axiomatically given subtyping relation is not new. Our observation expresses something more elementary and "syntax-free", however: The existence of a coercion between regular types, however specified, implies containment of the corresponding regular expressions. Note the direction of reasoning: from existence of coercion to containment.

Coinductive regular expression containment axiomatization with computational interpretation

We give a general coinductive axiomatization of regular expression containment and show how to interpret containment proofs computationally as string-preserving transformations on parse trees. Each rule in our axiomatization corresponds to a natural functional programming construct. Specifically, the coinduction rule corresponds to the principle of definition by recursion, where the side condition guarantees that the resulting function is total.

We show that the derivations of the axiomatizations by Salomaa [108], Kozen [76] and Grabmayer [52] can be coded as coercion judgements in our inference system. This provides a natural computational interpretation for their axiomatizations.

As far as we know, no previous regular expression *axiomatization* has explicitly been given a sound and complete computational interpretation, where *all* derivations are interpreted computationally. Sulzmann and Lu [113] come close,

however. They provide what can be considered the first coercion synthesis algorithm, implemented in an extension of Haskell. They show how to construct an explicit coercion for each valid regular expression containment by providing a computational interpretation of Antimirov's algorithm [12, 13] for deciding regular expression containment. They show that their treatment is sound [113, Lemma A.3], and state that it is complete. We observe that, being based on the construction of deterministic linear forms, their work can be thought of as implementing a proof search using Antimirov's algorithm in Grabmayer's axiomatization.

Parametric completeness

Let us define $E[X_1, \ldots, X_m] \leqslant F[X_1, \ldots, X_m]$ if the containment holds for all substitutions of X_i with (closed) regular expressions. Our axiomatization is not only complete, but *parametrically complete* for infinite alphabets:
If $E[X_1, \ldots, X_m] \leqslant F[X_1, \ldots, X_m]$ for all regular expressions $X_1, \ldots, X_m$ then there exists c such that $\vdash c : E[X_1, \ldots, X_m] \leqslant F[X_1, \ldots, X_m]$. As a consequence, a schematic axiom such as $E \times E^* \leqslant E^* \times E$ is *derivable*, not just admissible in our axiomatization: we can prove it once and use the *same* proof for all instances of E.

We observe that Kozen's axiomatization [76] is also parametrically complete, but neither Salomaa's [108] nor Grabmayer's [52] appear to be so: In Salomaa's case we need to make a case distinction as to whether the regular expression E substituted for X has the empty word property; and in Grabmayer's case the proofs use the derivatives of E, which are syntax dependent.

Application: Bit coding

We believe regular expressions as types with coercions have numerous applications in programming, both conceptually – how to *think* about regular expressions – and technically. We sketch one potential application: how *bit coding* can be used to compactly represent parse trees and thus strings. This can be thought of as a regular expression specific string representation that often can be compressed more than the original string. A thorough investigation of this and other applications requires separate treatment, however.

2.1.2 *Prerequisites*

We assume basic knowledge of regular expressions as in Hopcroft and Ullman [62], and denotational semantics as in Winskel [125].

2.1.3 *Notation and terminology*

$\mathcal{A}$ denotes an *alphabet*, a possibly infinite set of *symbols* $\{a_i\}_{i\in I}$. The *strings* over $\mathcal{A}$ is the set of finite sequences $\{s, t, \ldots\}$ with elements from $\mathcal{A}$. The *length* of a string s is denoted by $|s|_1$. The n-ary concatenation of $s_1, \ldots, s_n$ is denoted by their juxtaposition $s_1 \ldots s_n$; for $n = 0$ it denotes the empty string ϵ.

We use inl and inr as the tags distinguishing the elements of a disjoint sum of two sets such that
$X + Y = \{\mathrm{inl}\, v \mid v \in X\} \cup \{\mathrm{inr}\, w \mid w \in Y\}$. We treat recursive types *iso-recursively*, where $(\mathrm{fold}^{-1}, \mathrm{fold})$ denotes the isomorphism between a recursive type and its unrolling. In particular, we define the list type X^* by $\mu Y.1 + X \times Y$. The empty list $[]$ is an abbreviation for $\mathrm{fold}\,(\mathrm{inl}\,())$; and $\mathrm{cons}\,(x, y)$ stands for $\mathrm{fold}\,(\mathrm{inr}\,(x, y))$. The list notation $[x_1, \ldots, x_n]$ is syntactic sugar for $\mathrm{cons}\,(x_1, \ldots, \mathrm{cons}\,(x_n, []))$.

We say a unary predicate P *universally implies* another unary predicate Q if $\forall x.(P(x) \Rightarrow Q(x))$.

2.2 REGULAR EXPRESSIONS AS TYPES AND COERCIONS

In this section we show that a regular expression E can be interpreted as an ordinary *type* and regular expression containment as the existence of a *coercion* between such types. The elements of the types correspond to *proofs* of membership of strings in the regular language denoted by E, which in turn are the parse trees for E viewed as a right-regular *grammar*. A coercion then is any function that transforms parse trees without changing the underlying string.

Definition 9 (Regular expression)**.** *The set of* regular expressions $\mathrm{Reg}_{\mathcal{A}}$ *are defined by the following regular tree grammar:*

$$E, F, G, H ::= 0 \mid 1 \mid a \mid E + F \mid E \times F \mid E^*$$

where $a \in \mathcal{A}$.

In anticipation of our interpretation of regular expressions as types we write $\times$ instead of the more customary juxtaposition or $\cdot$ for concatenation. Our notational convention is that $*, \times, +$ bind in decreasing order; e.g. $a + a \times b$ stands for $a + (a \times b)$.

2.2.1 *Regular expressions as languages*

The *language interpretation* of $\mathrm{Reg}_{\mathcal{A}}$ maps regular expressions to *regular languages* [73]. This is also called the *standard interpretation* of $\mathrm{Reg}_{\mathcal{A}}$ since it is isomorphic to the free Kleene algebra over $\mathcal{A}$ [76].

Definition 10 (Language interpretation). *The* language $\mathcal{L}[\![E]\!]$ *is the set of strings compositionally defined by:*

$$\begin{array}{rclcrcl} \mathcal{L}[\![0]\!] & = & \varnothing & & \mathcal{L}[\![E+F]\!] & = & \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![1]\!] & = & \{\epsilon\} & & \mathcal{L}[\![E\times F]\!] & = & \mathcal{L}[\![E]\!] \cdot \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![a]\!] & = & \{a\} & & \mathcal{L}[\![E^*]\!] & = & \bigcup_{i\geqslant 0} (\mathcal{L}[\![E]\!])^i \end{array}$$

where $S \cdot T = \{st \mid s \in S \wedge t \in T\}$, $E^0 = \{\epsilon\}, E^{i+1} = E \cdot E^i$.

We write $\models s \in E$ *if* $s \in \mathcal{L}[\![E]\!]$; $\models E \leqslant F$ *if* $\mathcal{L}[\![E]\!] \subseteq \mathcal{L}[\![F]\!]$; *and* $\models E = F$ *if* $\mathcal{L}[\![E]\!] = \mathcal{L}[\![F]\!]$.

As expected, $\mathcal{L}[\![E^*]\!]$ is the set of all finite concatenations of strings from $\mathcal{L}[\![E]\!]$:

$$\mathcal{L}[\![E^*]\!] = \{s_1 \ldots s_n \mid n \geqslant 0 \wedge s_i \in \mathcal{L}[\![E]\!] \text{ for all } 1 \leqslant i \leqslant n\}.$$

Definition 11 (Constant part). *The* constant part $o(E)$ *of* E *is defined as* $o(E) = 1$ *if* $\epsilon \in \mathcal{L}[\![E]\!]$ *and* $o(E) = 0$ *otherwise.*

Definition 12 (Matching). *We say* s matches E *and write* $\vdash s \in E$ *if the statement* $s \in E$ *is derivable in the inference system in Figure 13a.*

Matching is sound and complete for membership testing:

Proposition 13. $\models s \in E$ *if and only if* $\vdash s \in E$.

The *derivation* of a matching statement $s \in E$ describes a *parse tree* for s under E understood as a regular grammar. This paper is about studying the parse trees, not just the regular language denoted by E. A sometimes unnoticed fact is that parse trees are in one-to-one correspondence with regular expressions interpreted as *types*; that is, all we need to do is interpret the regular expression constructors as type constructors and we obtain *exactly* the parse trees.

2.2.2 *Regular expressions as types*

Definition 14 (Type interpretation). *The* type interpretation $\mathcal{T}[\![.]\!]$ *compositionally maps a regular expression* E *to a set of structured values:*

$$\begin{array}{rclcrcl} \mathcal{T}[\![0]\!] = \varnothing & & & & \mathcal{T}[\![E+F]\!] & = & \mathcal{T}[\![E]\!] + \mathcal{T}[\![F]\!] \\ \mathcal{T}[\![1]\!] = \{()\} & & & & \mathcal{T}[\![E\times F]\!] & = & \mathcal{T}[\![E]\!] \times \mathcal{T}[\![F]\!] \\ \mathcal{T}[\![a]\!] = \{a\} & & & & \mathcal{T}[\![E^*]\!] & = & \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{T}[\![E]\!]\} \end{array}$$

We write $\models v : E$ *if* $v \in \mathcal{T}[\![E]\!]$.

Figure 13 Matching relation and type inhabitation

$$\frac{}{\epsilon \in 1} \qquad \frac{}{() : 1}$$

$$\frac{}{a \in a} \qquad \frac{}{a : a}$$

$$\frac{s \in E}{s \in E + F} \qquad \frac{v : E}{\mathrm{inl}\, v : E + F}$$

$$\frac{s \in F}{s \in E + F} \qquad \frac{w : F}{\mathrm{inr}\, w : E + F}$$

$$\frac{s \in E \qquad t \in F}{s\,t \in E \times F} \qquad \frac{v : E \qquad w : F}{(v, w) : E \times F}$$

$$\frac{s \in 1 + E \times E^*}{s \in E^*} \qquad \frac{v : 1 + E \times E^*}{\mathrm{fold}\, v : E^*}$$

a) Regular expression matching **b)** Type inhabitation

Note that this is the ordinary interpretation of the regular expression constructors as type constructors: 0 is the empty type, 1 the unit type, a (as a type) the singleton type $\{a\}$, $+$ the sum type constructor, $\times$ the product type constructor, and $.^*$ the list type constructor.

Definition 15 (Inhabitation)**.** *We say* v inhabits E *and write* $\vdash v : E$ *if the statement* $v : E$ *is derivable in the inference system in Figure 13b.*

Inhabitation is sound and complete for type membership:

Proposition 16. $\models v : E$ *if and only if* $\vdash v : E$.

By inspection of Figures 13a and 13b we can see that a value v such that $\vdash v : E$ corresponds to a unique derivation of $s \in E$ for a string s that is uniquely determined by *flattening* v.

Definition 17. *The* flattening *function* $\mathrm{flat}(.)$ *from values to strings is defined as follows:*

$$\begin{aligned} \mathrm{flat}(()) &= \epsilon & \mathrm{flat}(a) &= a \\ \mathrm{flat}(\mathrm{inl}\, v) &= \mathrm{flat}(v) & \mathrm{flat}(\mathrm{inr}\, w) &= \mathrm{flat}(w) \\ \mathrm{flat}((v, w)) &= \mathrm{flat}(v)\,\mathrm{flat}(w) & \mathrm{flat}(\mathrm{fold}\, v) &= \mathrm{flat}(v) \end{aligned}$$

In particular we have:

Theorem 18. $\mathcal{L}[\![E]\!] = \{\mathrm{flat}(v) \mid v \in \mathcal{T}[\![E]\!]\}$

2.2.3 *Regular expression containment as type coercion*

Since each regular expression can be thought of as an ordinary type whose elements are all the parse trees for all its strings under the language interpretation, we can characterize regular language containment as the problem of transforming parse trees under one regular expression into parse trees under the other regular expression.

Definition 19 (Coercion)**.** *A function* $f \in \mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]_\perp$ *is a* partial coercion from E to F *if* $\mathrm{flat}(v) = \mathrm{flat}(f(v))$ *for all* $v \in \mathcal{T}[\![E]\!]$ *where* $f(v) \neq \perp$*. It is a* total coercion *(or just* coercion*) if* $f(v) \neq \perp$ *for all* $v \in \mathcal{T}[\![E]\!]$*.*

We write $\mathcal{T}[\![E \leqslant F_\perp]\!]$ *for the set of partial coercions from* E *to* F*; and* $\mathcal{T}[\![E \leqslant F]\!]$ *for the set of total coercions from* E *to* F*.*

In other words, a coercion from E to F is a function that transforms every parse tree under E to a parse tree under F for the same underlying string. Clearly, if there exists a coercion from E to F then $\models E \leqslant F$: the coercion takes any membership proof of a string in $\mathcal{L}[\![E]\!]$ to a membership proof for the same string in $\mathcal{L}[\![F]\!]$. Conversely, if $\models E \leqslant F$, we can define a coercion from E to F by mapping any value $v : E$ to a value $w : F$ where $\mathrm{flat}(v) = \mathrm{flat}(w)$.

Theorem 20 (Containment by coercion)**.** $\models E \leqslant F$ *if and only if there exists a coercion from* E *to* F*.*

An immediate corollary is that two regular expressions are equivalent if and only if there is a pair of coercions between them:

Corollary 21 (Equivalence by coercion pairs)**.** $\models E = F$ *if and only if there exists a pair of coercions* (f, g) *such that* $f \in \mathcal{T}[\![E \leqslant F]\!]$ *and* $g \in \mathcal{T}[\![F \leqslant E]\!]$*.*

It may be tempting to expect such pairs to be isomorphisms; that is, $f \circ g = \mathrm{id}_{\mathcal{T}[\![F]\!]}$ and $g \circ f = \mathrm{id}_{\mathcal{T}[\![E]\!]}$). This is generally not the case, however: We have $\models a = a + a$, but there is no isomorphism between them, since there are two values for $a + a$ but only one value for a.

Theorem 20 provides a simple and amazingly useful method for proving regular expression containments by *functional programming*: Find a function from E to F (as *types*!) and make sure that it terminates, outputs each part of the input exactly once and in the same left-to-right order. The latter is usually easily checked when using pattern matching in the definition of the function.

Example 22. *We prove the* denesting *rule [77]* $\models (a + b)^* = a^* \times (b \times a^*)^*$*. In one direction, find a function* $f : (a + b)^* \to a^* \times (b \times a^*)^*$ *and make sure that it terminates,*

uses each part of the input exactly once and outputs them in the same left-to-right order as in the input.

$$\begin{aligned} f([]) &= ([], []) \\ f(\mathrm{inl}\, u :: \vec{z}) &= \textbf{let}\, (\vec{x}, \vec{y}) = f(\vec{z})\, \textbf{in}\, (u :: \vec{x}, \vec{y}) \\ f(\mathrm{inr}\, v :: \vec{z}) &= \textbf{let}\, (\vec{x}, \vec{y}) = f(\vec{z})\, \textbf{in}\, ([], (v, \vec{x}) :: \vec{y}) \end{aligned}$$

We can see that f *terminates since it is called recursively with smaller sized arguments, and the output contains the input components in the same left-to-right order. Consequently,* f *defines a coercion, and by Theorem 20 this constitutes a proof that the regular language* $\mathcal{L}[\![(a + b)^*]\!]$ *is contained in* $\mathcal{L}[\![a^* \times (b \times a^*)^*]\!]$*. The other direction is similar.*

In this example we defined an element of the function space $\mathcal{T}[\![E]\!] \to \mathcal{T}[\![F]\!]_\perp$ and then verified manually that it belongs to the subspace $\mathcal{T}[\![E \leqslant F]\!]$. The following section is about designing a *language* of functions each of which is guaranteed to be a coercion (soundness) and that furthermore is expressive enough so that it contains a term denoting a coercion from E to F whenever $\models E \leqslant F$ (completeness).

2.3 DECLARATIVE COINDUCTIVE AXIOMATIZATION

At the core of all axiomatizations of regular expression equivalence are the axiomatization of product ($\times$), sum ($+$), empty (0) and unit (1) as the free idempotent semiring over $\mathcal{A}$. See Figures 14 and 15. We add the familiar *fold/unfold* axiom for Kleene-star in Figure 16, which models that E^* is a fixed point of $X = 1 + E \times X$. Let us call the resulting inference system *weak equivalence*. It is a sound, but incomplete axiomatization of regular expression equivalence. In particular, we have $\models (a + 1)^* = a^*$, but they are not weakly equivalent [108, Remark 4].

Intuitively, this is because weak equivalence does not allow invoking *recursively* what we want to prove. The basic idea in our axiomatization is to add recursion by way of a general *finitary coinduction* rule

$$\frac{\begin{array}{c}[E = F] \\ \vdots \\ E = F\end{array}}{E = F}\ (*)$$

Here $[E = F]$ is a *hypothetical assumption*: It may be used an arbitrary number of times in deriving the premise, but is *discharged* when applying the inference step.

Since the premise is the same as the conclusion, without a side condition $(*)$ restricting its applicability this rule is blatantly unsound: We could simply satisfy the premise by immediately concluding $E = F$ from the hypothetical assumption. By the coinduction rule $E = F$ for arbitrary E, F would be derivable.

Figure 14 Axioms for idempotent semirings

$$
\begin{aligned}
E + (F + G) &= (E + F) + G && (2.1)\\
E + F &= F + E && (2.2)\\
E + 0 &= E && (2.3)\\
E + E &= E && (2.4)\\
E \times (F \times G) &= (E \times F) \times G && (2.5)\\
1 \times E &= E && (2.6)\\
E \times 1 &= E && (2.7)\\
E \times (F + G) &= (E \times F) + (E \times G) && (2.8)\\
(E + F) \times G &= (E \times G) + (F \times G) && (2.9)\\
0 \times E &= 0 && (2.10)\\
E \times 0 &= 0 && (2.11)
\end{aligned}
$$

The key idea of this paper is to make the side condition not a property of the premise, but of the *derivation* of the premise. To this end we switch from axiomatizing equivalence to axiomatizing containment and equip our inference system with names for the rules, arriving at a type-theoretic formulation with explicit *proof terms*. These proof terms can be computationally interpreted as *coercions* as defined in Section 2.2.

The coinduction rule then suggestively reads

$$
\frac{\begin{array}{c}[f : E \leqslant F]\\ \vdots \\ c : E \leqslant F\end{array}}{\mathrm{fix} f.\, c : E \leqslant F}\ (*)
$$

as in Brandt and Henglein [23, Section 4.4], where the side condition is a syntactic condition specific to recursive subtyping. The *computational interpretation* of $\mathrm{fix} f.\, c$ is the *recursively defined* partial coercion f such that $f = c$ where c may contain free occurrences of f. For soundness all we need is for the coercion to be total, that is terminate on all inputs. This leads us to the side condition in its most general form:

> The computational interpretation of $\mathrm{fix} f.\, c$ must be *total*.

Unfortunately, totality turns out to be undecidable, and we present efficiently checkable syntactic conditions that entail totality and yet are expressive enough to admit completeness.

Figure 15 Rules of equality

$$\mathsf{E} = \mathsf{E} \qquad \frac{\mathsf{E} = \mathsf{F}}{\mathsf{F} = \mathsf{E}} \qquad \frac{\mathsf{E} = \mathsf{F} \quad \mathsf{F} = \mathsf{G}}{\mathsf{E} = \mathsf{G}}$$

$$\frac{\mathsf{E} = \mathsf{G} \quad \mathsf{F} = \mathsf{H}}{\mathsf{E} + \mathsf{F} = \mathsf{G} + \mathsf{H}} \qquad \frac{\mathsf{E} = \mathsf{G} \quad \mathsf{F} = \mathsf{H}}{\mathsf{E} \times \mathsf{F} = \mathsf{G} \times \mathsf{H}}$$

2.3.1 *Axiomatization*

Consider the coercion inference system in Figure 17. Each axiom of the form $\Gamma \vdash p : \mathsf{E} = \mathsf{F}$ is a short-hand for two containment axioms: $\Gamma \vdash p : \mathsf{E} \leqslant \mathsf{F}$ and $\Gamma \vdash p^{-1} : \mathsf{F} \leqslant \mathsf{E}$.

Definition 23 (Coercion judgement)**.** *Let* Γ *be a sequence of* coercion assumptions *of the form* $f : \mathsf{E} \leqslant \mathsf{F}$*. A* coercion judgement *is a statement of the form* $\Gamma \vdash c : \mathsf{E} \leqslant \mathsf{F}$ *that is derivable in the inference system of Figure 17.*

If Γ is empty we simply omit it and write $\vdash c : \mathsf{E} \leqslant \mathsf{F}$.

By induction on its derivation, a coercion judgement $f_1 : \mathsf{E}_1 \leqslant \mathsf{F}_1, \ldots, f_n : \mathsf{E}_n \leqslant \mathsf{F}_n \vdash c : \mathsf{E} \leqslant \mathsf{F}$ can be interpreted coherently as a continuous function $\mathcal{F}[\![\Gamma \vdash c : \mathsf{E} \leqslant \mathsf{F}]\!] : \mathcal{T}[\![\mathsf{E}_1 \leqslant \mathsf{F}_{1\perp}]\!] \times \ldots \times \mathcal{T}[\![\mathsf{E}_n \leqslant \mathsf{F}_{n\perp}]\!] \to \mathcal{T}[\![\mathsf{E} \leqslant \mathsf{F}_\perp]\!]$, which is specified by the equations in Figure 18. For example, the clauses for retag should be understood as

$$\begin{aligned}&\mathcal{F}[\![\Gamma \vdash \mathrm{retag} : \mathsf{E} + \mathsf{F} \leqslant \mathsf{F} + \mathsf{E}]\!](f_1, \ldots, f_n) = \\ &\quad \lambda x.\mathbf{case}\, x\, \mathbf{of}\, \mathrm{inl}\, v \Rightarrow \mathrm{inr}\, v \mid \mathrm{inr}\, v \Rightarrow \mathrm{inl}\, v.\end{aligned}$$

The interpretation of $\mathrm{fix} f.c$ is defined to be the least fixed point of a continuous function on $\mathcal{T}[\![\mathsf{E} \leqslant \mathsf{F}_\perp]\!]$, which always exists since $\mathcal{T}[\![\mathsf{E} \leqslant \mathsf{F}_\perp]\!]$ is empty or a cpo with bottom. The interpretation of $\mathrm{abortL}, \mathrm{abortR}, \mathrm{abortL}^{-1}, \mathrm{abortR}^{-1}$ is the empty function since the type interpretation of their domain is empty. To summarize:

Definition 24 (Computational interpretation)**.**
The computational interpretation

$$\mathcal{F}[\![f_1 : \mathsf{E}_1 \leqslant \mathsf{F}_1, \ldots, f_n : \mathsf{E}_n \leqslant \mathsf{F}_n \vdash c : \mathsf{E} \leqslant \mathsf{F}]\!]$$

Figure 16 Fold/unfold rule for Kleene-star

$$\mathsf{E}^* \quad = \quad 1 + \mathsf{E} \times \mathsf{E}^* \tag{2.12}$$

Figure 17 Declarative coercion inference system for regular expressions as types. With suitable side conditions for the coinduction rule this is sound and complete for regular expression containment. See Sections 2.3.2 and 2.3.3 for side conditions.

$$
\begin{array}{lllrcl}
\Gamma & \vdash & \text{shuffle} & : & E + (F + G) & = & (E + F) + G \\
\Gamma & \vdash & \text{retag} & : & E + F & = & F + E \\
\Gamma & \vdash & \text{untagL} & : & 0 + F & = & F \\
\Gamma & \vdash & \text{untag} & : & E + E & \leqslant & E \\
\Gamma & \vdash & \text{tagL} & : & E & \leqslant & E + F \\
\Gamma & \vdash & \text{assoc} & : & E \times (F \times G) & = & (E \times F) \times G \\
\Gamma & \vdash & \text{swap} & : & E \times 1 & = & 1 \times E \\
\Gamma & \vdash & \text{proj} & : & 1 \times E & = & E \\
\Gamma & \vdash & \text{abortR} & : & E \times 0 & = & 0 \\
\Gamma & \vdash & \text{abortL} & : & 0 \times E & = & 0 \\
\Gamma & \vdash & \text{distL} & : & E \times (F + G) & = & (E \times F) + (E \times G) \\
\Gamma & \vdash & \text{distR} & : & (E + F) \times G & = & (E \times G) + (F \times G) \\
\Gamma & \vdash & \text{wrap} & : & 1 + E \times E^* & = & E^* \\
\Gamma & \vdash & \text{id} & : & E & = & E
\end{array}
$$

$$\frac{\Gamma \vdash c : E \leqslant E' \qquad \Gamma \vdash d : E' \leqslant E''}{\Gamma \vdash c; d : E \leqslant E''}$$

$$\frac{\Gamma \vdash c : E \leqslant E' \qquad \Gamma \vdash d : F \leqslant F'}{\Gamma \vdash c + d : E + F \leqslant E' + F'}$$

$$\frac{\Gamma \vdash c : E \leqslant E' \qquad \Gamma \vdash d : F \leqslant F'}{\Gamma \vdash c \times d : E \times F \leqslant E' \times F'}$$

$$\Gamma, f : E \leqslant F, \Gamma' \vdash f : E \leqslant F$$

$$\frac{\Gamma, f : E \leqslant F \vdash c : E \leqslant F}{\Gamma \vdash \text{fix} f.c : E \leqslant F} \quad \text{(coinduction rule)}$$

of a coercion judgement is the continuous function that maps partial coercions from E_i *to* F_i *bound to the* f_i *to a partial coercion from* E *to* F *satisfying the equations of Figure 18.*

We can interpret all computation judgements, but without a side condition controlling the use of the coinduction rule, the coercion inference system is unsound for deducing regular expression containments. To wit, we can trivially derive $\vdash \mathsf{fixf.f} : E \leqslant F$ for any E, F. We might hope that a simple *guarding rule* would ensure soundness.

Definition 25 (Left-guarded)**.** *Let* $\Gamma \vdash \mathsf{fixf.c} : E \leqslant F$ *be a coercion judgement. We say an occurrence of* f *in* c *is* left-guarded by d *if* c *contains a subterm of the form* $d \times d'$ *and* f *occurs in* d'*. We call* $\mathsf{fixf.c}$ left-guarded *if for each occurrence of* f *there is a* d *that left-guards* f*.*

Left-guardedness is not sufficient for soundness, however. Consider

$$\vdash \mathsf{fixf.}(\mathsf{proj}^{-1} ; (\mathsf{id}_1 \times f) ; \mathsf{proj}) : E \leqslant F,$$

which is derivable for all E and F. (For emphasis, we have annotated id with a subscript indicating which regular expression it operates on.) Computationally, this coercion judgement does not terminate on any input. This is an instructive case: It contains *both* a proj^{-1} coercion *and* an f that is left-guarded "only" by (a coercion operating on) a regular expression whose language contains the empty string, in this case 1.

2.3.2 *Soundness*

We have seen that, without a side condition on the coinduction rule, the coercion inference system is unsound for deducing regular expression containments. The key idea now is this: Impose a side condition that guarantees that the coercion in the conclusion of the coinduction rule is total. Since all other rules preserve totality of coercions, this yields a sound axiomatization of regular expression containment by Theorem 20. Since our coercions may contain free variables, we need to generalize totality to second-order coercions:

Definition 26 (Hereditary totality)**.** *We say coercion judgement* $\Gamma \vdash c : E \leqslant F$ *for* $\Gamma = f_1 : E_1 \leqslant F_1, \ldots, f_n : E_n \leqslant F_n$ *is* hereditarily total *if* $\mathcal{F}[\![\Gamma \vdash c : E \leqslant F]\!](f_1, \ldots, f_n)$ *is total whenever* f_i *is a total coercion from* E_i *to* F_i *for all* $i = 1, \ldots, n$*.*

We are now ready to define sound restrictions of the coercion inference system. Instead of formulating a specific side condition, we parameterize over side conditions for the coinduction rule to express, generally, what is necessary for such a side condition to guarantee soundness.

Figure 18 Computational interpretation of coercions

$$
\begin{array}{lcl}
\mathrm{shuffle}(\mathrm{inl}\, v) & = & \mathrm{inl}\,(\mathrm{inl}\, v) \\
\mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inl}\, v)) & = & \mathrm{inl}\,(\mathrm{inr}\, v) \\
\mathrm{shuffle}(\mathrm{inr}\,(\mathrm{inr}\, v)) & = & \mathrm{inr}\, v \\
\mathrm{shuffle}^{-1}(\mathrm{inl}\,(\mathrm{inl}\, v)) & = & \mathrm{inl}\, v \\
\mathrm{shuffle}^{-1}(\mathrm{inl}\,(\mathrm{inr}\, v)) & = & \mathrm{inr}\,(\mathrm{inl}\, v) \\
\mathrm{shuffle}^{-1}(\mathrm{inr}\, v) & = & \mathrm{inr}\,(\mathrm{inr}\, v) \\
\mathrm{retag}(\mathrm{inl}\, v) & = & \mathrm{inr}\, v \\
\mathrm{retag}(\mathrm{inr}\, v) & = & \mathrm{inl}\, v \\
\mathrm{retag}^{-1} & = & \mathrm{retag} \\
\mathrm{untagL}\,(\mathrm{inr}\, v) & = & v \\
\mathrm{untag}\,(\mathrm{inl}\, v) & = & v \\
\mathrm{untag}\,(\mathrm{inr}\, v) & = & v \\
\mathrm{tagL}\,(v) & = & \mathrm{inl}\, v \\
\mathrm{assoc}(v,(w,x)) & = & ((v,w),x) \\
\mathrm{assoc}^{-1}((v,w),x) & = & (v,(w,x)) \\
\mathrm{swap}(v,()) & = & ((),v) \\
\mathrm{swap}^{-1}((),v) & = & (v,()) \\
\mathrm{proj}((),w) & = & w \\
\mathrm{proj}^{-1}(w) & = & ((),w) \\
\mathrm{distL}(v,\mathrm{inl}\, w) & = & \mathrm{inl}\,(v,w) \\
\mathrm{distL}(v,\mathrm{inr}\, x) & = & \mathrm{inr}\,(v,x) \\
\mathrm{distL}^{-1}(\mathrm{inl}\,(v,w)) & = & (v,\mathrm{inl}\, w) \\
\mathrm{distL}^{-1}(\mathrm{inr}\,(v,x)) & = & (v,\mathrm{inr}\, x) \\
\mathrm{distR}(\mathrm{inl}\, v,w) & = & \mathrm{inl}\,(v,w) \\
\mathrm{distR}(\mathrm{inr}\, v,x) & = & \mathrm{inr}\,(v,x) \\
\mathrm{distR}^{-1}(\mathrm{inl}\,(v,w)) & = & (\mathrm{inl}\, v,w) \\
\mathrm{distR}^{-1}(\mathrm{inr}\,(v,x)) & = & (\mathrm{inr}\, v,x) \\
\mathrm{wrap}\,(v) & = & \mathrm{fold}\, v \\
\mathrm{wrap}^{-1}(v) & = & \mathrm{fold}^{-1}\, v \\
\mathrm{id}(v) & = & v \\
\mathrm{id}^{-1} & = & \mathrm{id} \\
(c;d)(v) & = & d(c(v)) \\
(c+d)(\mathrm{inl}\, v) & = & \mathrm{inl}\,(c(v)) \\
(c+d)(\mathrm{inr}\, w) & = & \mathrm{inr}\,(d(w)) \\
(c\times d)(v,w) & = & (c(v),d(w)) \\
(\mathrm{fix} f.c)(v) & = & c[\mathrm{fix} f.c/f](v)
\end{array}
$$

Definition 27 (Coercion inference system with side condition)**.** *Consider the coercion inference system of Figure 17 where the coinduction rule is equipped with a side condition* P*, a predicate on the coercion judgement in the conclusion:*

$$\frac{\Gamma, f : E \leqslant F \vdash c : E \leqslant F}{\Gamma \vdash \mathrm{fix} f.c : E \leqslant F} \qquad (\mathsf{P}(\Gamma \vdash \mathrm{fix} f.c : E \leqslant F)).$$

We write $\Gamma \vdash_{\mathsf{P}} c : E \leqslant F$*, if each application of the coinduction rule in the derivation of* $\Gamma \vdash c : E \leqslant F$ *satisfies* P*.*

We arrive at the Master Soundness Theorem, which provides a general criterion for sound side conditions:

Theorem 28 (Soundness)**.** *Let* P *be any predicate on coercion judgements that universally implies hereditary totality.*

Then $\vdash d : E \leqslant F$ *implies* $\models E \leqslant F$ *for all* d, E, F*.*

This theorem shows that hereditary totality is an "upper bound" for how liberal the side condition can be without the risk of losing sound computational interpretation of a regular expression containment proof as a coercion. Interestingly, allowing *partial* coercions does not *necessarily* make the resulting inference system unsound for proving regular expression containment. If we define the side condition

$$\mathsf{P}^{\mathsf{t}}(\Gamma \vdash \mathrm{fix} f.c : E \leqslant F) \iff \models E \leqslant F,$$

the resulting inference system is trivially sound and complete since $\vdash \mathrm{fix} f.f : E \leqslant F$ is derivable for those E, F such that $\models E \leqslant F$. Clearly, $\mathcal{F}[\![\vdash \mathrm{fix} f.f : E \leqslant F]\!]$ is computationally completely useless, however: it never terminates.

Unfortunately, hereditary totality itself is undecidable even for the restricted language of coercions denotable by coercion judgements:[3]

Theorem 29. *Whether or not* $\Gamma \vdash c : E \leqslant F$ *is hereditarily total is undecidable.*

Proof. Even totality of $\vdash c : 1 \leqslant 1$ is undecidable. This follows from the undecidability of $\vdash c : 1^* \times 1^* \leqslant 1^* \times 1^*$, which in turn follows from encoding Minsky machines (2-register machines) as closed coercion judgements, using a unary coding of natural numbers. □

This makes hereditary totality inapplicable as a conventional side condition in an axiomatization, where valid instances of an inference rule are expected to be decidable. Below we provide polynomial-time decidable side conditions that are sufficient to encode existing derivations in previous axiomatizations (see Section 2.3.3). In each case their soundness follows from application of Theorem 28. In this sense, hereditary totality can be considered the "mother of all side conditions", even though it itself is "too extensional" to be used as a conventional side condition.

3 Proved by Eijiro Sumii, Yasuhiko Minamide, Naoki Kobayashi, Atsushi Igarashi and Fritz Henglein at the IFIP TC 2 Working Group 2.8 meeting at Shirahama, Japan, April 11-16, 2010

Definition 30 (Syntactic side conditions S_i)**.** *Define predicates* S_1, S_2, S_3 *and* S_4 *on coercion judgements of the form* $\Gamma \vdash \mathrm{fix}f.c : E \leqslant F$ *as follows:*

- $S_1(\Gamma \vdash \mathrm{fix}f.c : E \leqslant F)$ *if and only if each occurrence of* f *in* c *is left-guarded by a* d *where* $\Gamma, \ldots \vdash d : E' \leqslant F'$ *is the coercion judgement for* d *occurring in the derivation of* $\Gamma \vdash \mathrm{fix}f.c : E \leqslant F$ *and* $o(E') = 0$ *(from Definition 11).*
- $S_2(\Gamma \vdash \mathrm{fix}f.c : E \leqslant F)$ *if and only if each occurrence of* f *in* c *is left-guarded and for each subterm of the form* $c_1; c_2$ *in* c *at least one of the following conditions is satisfied:*
  - c_1 *is closed and* proj^{-1}*-free;*
  - c_2 *is closed.*
- $S_3(\Gamma \vdash \mathrm{fix}f.c : E \leqslant F)$ *if* c *is of the form* $\mathrm{wrap}^{-1}; (\mathrm{id} + \mathrm{id} \times f); d$ *where* d *is closed.*
- $S_4 = S_1 \vee S_3$.

It is easy to see that S_1, S_2, S_3 and S_4 are polynomial-time checkable. They furthermore imply hereditary totality:

Lemma 31 (Hereditary totality for S_i)**.** *Let* $\Gamma \vdash \mathrm{fix}f.c : E \leqslant F$ *such that* $S_i(\Gamma \vdash \mathrm{fix}f.c : E \leqslant F)$ *with* $i \in \{1, 2, 3, 4\}$.

Then $\Gamma \vdash \mathrm{fix}f.c : E \leqslant F$ *is hereditarily total.*

Side condition S_3 is a special case of S_2. The case of S_4 follows from S_1 and S_3. We have formulated S_3 separately since S_4 is sufficient to code all derivations in Salomaa's and Grabmayer's axiomatizations. S_2 by itself, without S_1, is sufficient for Kozen's axiomatization.

Proof. (Idea) The general idea behind the side conditions S_1, S_2 is that they ensure that every recursive call f in the body c of a recursively defined coercion $\mathrm{fix}f.c$ is called with an argument whose *size* is properly smaller than the size of the original call. The difference between the two conditions is the definition of size in each case.

Consider S_1. Define the 0*-size* $|v|_0$ of a value by $|v|_0 = |\mathrm{flat}(v)|_1$; that is, it is the length of the underlying string. Values containing $()$ may be of 0-size 0, e.g. $|((), ())|_0 = 0$, and the size of a component of a pair may be the same as the size of the pair: $|((), v)|_0 = |v|_0$. Consider a call of $\mathrm{fix}f.c$ to a value v of 0-size n. The predicate S_1 ensures that all recursive calls to f in c are only applied to a value constructed from the second component of some pair, where the first component has size at least 1. Since coercions never increase the size this guarantees that the recursive call is applied to a value of 0-size at most $n - 1$.

Now consider S_2. Define the 1-size $|v|_1$ of v as follows:

$$\begin{array}{rclcrcl} |()|_1 & = & 1 & & |a|_1 & = & 1 \\ |\mathrm{inl}\, v|_1 & = & |v|_1 & & |\mathrm{inr}\, v|_1 & = & |v|_1 \\ |\mathrm{fold}\, v|_1 & = & |v|_1 & & |(v,w)|_1 & = & |v|_1 + |w|_1 \end{array}$$

Note that if we had defined $|()|_1$ to be 0 then this would be just the 0-size (hence our terminology).

The idea for ensuring termination of a recursively defined coercion is the same as before, but for 1-size instead of 0-size. With 1-size we have the important property that each component of a pair is *properly* smaller than the pair, in particular $|v_2|_1 < |(v_1, v_2)|_1$ for *all* v_1. We say a coercion c is *nonexpansive* if $|c(v)|_1 \leqslant |v|_1$. All primitive coercions *except for* proj^{-1} are nonexpansive, and the inference rules preserve nonexpansiveness. Now, S_2 guarantees that each recursive call is applied to an argument of size properly smaller than the original call. Informally, this is because S_2 guarantees that a recursive call of f is never applied to a value (constructed from) the *output* (return value) of a proj^{-1}-call. □

From Theorem 28, Lemma 31 and Theorem 20 we obtain:

Corollary 32 (Soundness for side conditions S_i)**.** *Let* S_1, S_2, S_3, S_4 *as in Definition 30,* $i \in \{1,2,3,4\}$.

Then $\vdash_{S_i} d : E \leqslant F$ *implies* $\models E \leqslant F$.

2.3.3 *Completeness*

We show now how to code derivations in Salomaa's, Kozen's and Grabmayer's axiomatizations of regular expression equivalence in our coercion inference system (Figure 17) with side condition S_4 (Salomaa, Grabmayer) or S_2 (Kozen). This provides a computational interpretation for each of these systems. Furthermore, it implies that coercion axiomatization with either S_2 or S_4 is complete. More precisely, we encode every derivation of $\vdash E = F$ as a pair of coercion judgements $\vdash c : E \leqslant F$ and $\vdash d : F \leqslant E$, respectively, which provides a computational interpretation of a regular expression equivalence as a pair of coercions.

Even though they are for regular expression *equivalence*, these codings also provide completeness of our coercion axiomatization for regular expression *containment*. Assume $\models E \leqslant F$. This holds if and only if $\models E + F = F$. By completeness of the regular expression equivalence axiomatizations, $E + F = F$ is derivable, and we can construct a coercion judgement of $\vdash c : E + F \leqslant F$. Composes with tagL this yields $\vdash \mathrm{tagL}\,; c : E \leqslant F$, and we are done.

Theorem 33 (Completeness)**.** *Let* P *be either* S_2 *or* S_4. *Then* $\models E \leqslant F$ *implies* $\vdash_P E \leqslant F$

Figure 19 Salomaa's rules for axiomatization F_1

$$\frac{E = F}{E^* = F^*} \qquad E^* = (1 + E)^* \qquad \frac{E = F \times E + G}{E = F^* \times G}\ (\text{IF } o(F) = 0)$$

It follows that any side condition "between" S_2 or S_4 on the one hand and hereditary totality on the other hand yields a sound and complete coercion axiomatization of regular expression containment.

Corollary 34 (Soundness and completeness)**.** *Let* P *be such that either* S_2 *or* S_3 *universally implies* P*, and* P *universally implies hereditary totality. Then* $\vdash_P c : E \leqslant F$ *if and only if* $\models E \leqslant F$*.*

Whereas hereditary totality is the natural "upper" bound we suspect that there are natural weaker "lower" bounds than S_2 and S_4.

Salomaa

Salomaa's System F_1 [108] arises from adding the rules of Figure 19 to the axiomatization of weak equivalence (Figures 14, 15 and 16).[4] The side condition of the inference rule in Figure 19 is called the "no empty word property".

To be precise, we prove by induction on derivations of $E = F$ in System F_1 that there exist coercion judgements $\vdash_{S_4} c : E \leqslant F$ and $\vdash_{S_4} d : F \leqslant E$. This is straightforward for the weak equivalence rules. We thus concentrate on the rules in Figure 19.

Consider $\dfrac{E = F}{E^* = F^*}$. By induction hypothesis there exist $\vdash_{S_4} c : E \leqslant F$ and $\vdash_{S_4} d : F \leqslant E$. We reason as follows: Assume $E^* \leqslant F^*$ and call this assumption f.

$$\begin{array}{rcll} E^* & \leqslant & (1 + E \times E^*) & \text{by wrap}^{-1} \\ & \leqslant & (1 + F \times F^*) & \text{by id} + c \times f \\ & \leqslant & F^* & \text{by wrap} \end{array}$$

This shows that

$$f : E^* \leqslant F^* \vdash_{S_4} (\text{wrap}^{-1}; \text{id} + c \times f; \text{wrap}) : E^* \leqslant F^*.$$

Note that $\vdash \text{fix} f.(\text{wrap}^{-1}; \text{id} + c \times f; \text{wrap}) : E^* \leqslant F^*$ satisfies side condition S_3 and thus S_4. Its computational interpretation is the map-function on lists. With S_4

4 Technically, this is the "left-handed" dual due to Grabmayer [52] to Salomaa's original "right-handed" formulation where the fold-unfold rule for Kleene star is axiomatized as $E^* = 1 + E^* \times E$.

satisfied we can apply the coinduction rule to conclude $\vdash_{S_4} \mathrm{fix} f.(\mathrm{wrap}^{-1}; \mathrm{id} + c \times f; \mathrm{wrap}) : E^* \leqslant F^*$. Similarly, we get $\vdash_{S_4} \mathrm{fix} g.(\mathrm{wrap}^{-1}; \mathrm{id} + d \times g; \mathrm{wrap}) : F^* \leqslant E^*$.

Consider $E^* = (1 + E)^*$. The case $E^* \leqslant (1 + E)^*$ follows from the rule above since $E \leqslant (1 + E)$. For the converse containment assume $f : (1 + E)^* \leqslant E^*$. We have:

$$\begin{array}{rcll}
(1 + E)^* & \leqslant & 1 + (1 + E) \times (1 + E)^* & \text{by } \mathrm{wrap}^{-1} \\
& \leqslant & 1 + (1 + E) \times E^* & \text{by } f \\
& \leqslant & 1 + 1 \times E^* + E \times E^* & \text{by distR} \\
& \leqslant & 1 + E^* + E \times E^* & \text{by proj} \\
& \leqslant & 1 + E \times E^* + E^* & \text{by retag} \\
& \leqslant & E^* + E^* & \text{by wrap} \\
& \leqslant & E^* & \text{by untag}
\end{array}$$

We are a bit informal here: We leave associativity, congruence and identity steps implicit. Let us consider $\vdash \mathrm{fix} f.c : (1 + E)^* \leqslant E^*$ now. Without displaying c in full detail, from the derivation above we can see that f satisfies side condition S_3 and thus S_4, and we can conclude $\vdash_{S_4} \mathrm{fix} f.c : (1 + E)^* \leqslant E^*$ by the coinduction rule. Operationally, $\mathcal{F}[\![\vdash \mathrm{fix} f.c : (1 + E)^* \leqslant E^*]\!]$ traverses its input list of type $\mathcal{T}[\![(1 + E)^*]\!]$, removes all occurrences of $\mathrm{inl}\,()$ and returns the v's for each $\mathrm{inr}\, v$ in the input.

Finally, consider $\dfrac{E = F \times E + G}{E = F^* \times G}$ [if $o(F) = 0$].

Our induction hypothesis is $\vdash_{S_4} c_1 : E \leqslant F \times E + G$ and $\vdash_{S_4} d_1 : F \times E + G \leqslant E$. Let us first consider $F^* \times G \leqslant E$. Assume $f : F^* \times G \leqslant E$, and we can calculate $d_2 : F^* \times G \leqslant E$ as follows:

$$\begin{array}{rcll}
F^* \times G & \leqslant & (1 + F \times F^*) \times G & \text{by } \mathrm{wrap}^{-1} \\
& \leqslant & 1 \times G + F \times F^* \times G & \text{by distR} \\
& \leqslant & G + F \times F^* \times G & \text{by proj} \\
& \leqslant & G + F \times E & \text{by } f \\
& \leqslant & F \times E + G & \text{by retag} \\
& \leqslant & E & \text{by } d_1
\end{array}$$

We can see that $\vdash \mathrm{fix} f.d_2 : F^* \times G \leqslant E$ satisfies S_2 and, since $o(F) = 0$, also S_1 and thus S_4. We can thus conclude $\vdash_{S_4} \mathrm{fix} f.d_2 : F^* \times G \leqslant E$ by the coinduction rule. Observe that $\vdash \mathrm{fix} f.d_2 : F^* \times G \leqslant E$ is hereditarily total, whether or not $o(F) = 0$, since S_2 is also satisfied.

Figure 20 Kozen's rules for axiomatization of Kleene Algebras

$$1 + (E^* \times E) \leqslant E^* \qquad \frac{E \times F \leqslant F}{E^* \times F \leqslant F} \qquad \frac{E \times F \leqslant E}{E \times F^* \leqslant E}$$

For the other direction, assume $\vdash_{S_4} g : E \leqslant F^* \times G$, and we can calculate $c_2 : E \leqslant F^* \times G$ essentially in the reverse direction to the above calculation.

$$\begin{array}{rcll} E & \leqslant & F \times E + G & \text{by } c_1 \\ & \leqslant & G + F \times E & \text{by retag} \\ & \leqslant & G + F \times F^* \times G & \text{by } g \\ & \leqslant & 1 \times G + F \times F^* \times G & \text{by proj}^{-1} \\ & \leqslant & (1 + F \times F^*) \times G & \text{by distR}^{-1} \\ & \leqslant & F^* \times G & \text{by wrap} \end{array}$$

Here, the coercion judgement $\vdash \mathrm{fix} g.c_2 : E \leqslant F^* \times G$ may computationally be nonterminating: Choose, e.g., $c_1 = \mathrm{proj}^{-1}; \mathrm{tagL} : E \leqslant 1 \times E + 0$. For $o(F) = 0$, however, $\vdash \mathrm{fix} g.c_2 : E \leqslant F^* \times G$ satisfies side condition S_1 and thus S_4; in particular, it always terminates. We can conclude $\vdash_{S_4} \mathrm{fix} g.c_2 : E \leqslant F \times E + G$ by the coinduction rule.

Kozen

Kozen [76] has shown that adding the rules in Figure 20 to weak equivalence is sound and complete for regular expression equivalence. Formally, a containment $E \leqslant F$ in his axiomatization is an abbreviation for $E + F = F$. We show now that all derivations in his system can be coded as coercion judgements with side condition S_2.

Consider $1 + (E^* \times E) \leqslant E^*$. It is sufficient to construct a coercion judgement $\vdash_{S_2} c : E^* \times E \leqslant E \times E^*$, since we then have $\vdash_{S_2} \mathrm{id} + c; \mathrm{wrap} : 1 \times E^* \times E \leqslant E^*$, as desired.

Assume $f : E^* \times E \leqslant E \times E^*$. We can calculate $c : E^* \times E \leqslant E \times E^*$ as follows:

$$\begin{array}{rcll} E^* \times E & \leqslant & (1 + E \times E^*) \times E & \text{by wrap}^{-1} \\ & \leqslant & 1 \times E + E \times E^* \times E & \text{by distR} \\ & \leqslant & 1 \times E + E \times E \times E^* & \text{by } f \\ & \leqslant & E \times 1 + E \times E \times E^* & \text{by swap} \\ & \leqslant & E \times (1 + E \times E^*) & \text{by distL}^{-1} \\ & \leqslant & E \times E^* & \text{by wrap} \end{array}$$

Writing c explicitly, we have

$$\begin{aligned} c = {} & (\mathrm{wrap}^{-1} \times \mathrm{id}); \mathrm{distR}; \\ & \mathrm{swap} + (\mathrm{assoc}^{-1}; \mathrm{id} \times f); \\ & \mathrm{distL}^{-1}; \mathrm{id} \times \mathrm{wrap} \end{aligned}$$

Observe that $\vdash \mathrm{fix} f.c : E^* \times E \leqslant E \times E^*$ satisfies side condition S_2, and we can conclude $\vdash_{S_2} \mathrm{fix} f.c : E^* \times E \leqslant E \times E^*$ by the coinduction rule.

Consider the rule $\dfrac{E \times F \leqslant F}{E^* \times F \leqslant F}$. Our induction hypothesis is that there exists $\vdash_{S_2} d : E \times F \leqslant F$. Assume $f : E^* \times F \leqslant F$, and we calculate $c : E^* \times F \leqslant F$ as follows:

$$\begin{array}{lcll} E^* \times F & \leqslant & (1 + E \times E^*) \times F & \text{by } \mathrm{wrap}^{-1} \\ & \leqslant & 1 \times F + E \times E^* \times F & \text{by distR} \\ & \leqslant & 1 \times F + E \times F & \text{by } f \\ & \leqslant & F + E \times F & \text{by proj} \\ & \leqslant & F + F & \text{by } d \\ & \leqslant & F & \text{by untag} \end{array}$$

Note that $\vdash \mathrm{fix} f.c : E^* \times F \leqslant F$ satisfies side condition S_2, and we can apply the coinduction rule to conclude $\vdash_{S_2} \mathrm{fix} f.c : E^* \times F \leqslant F$.

The rule $\dfrac{E \times F \leqslant E}{E \times F^* \leqslant E}$ is similar to the previous rule, with an additional step involving $E^* \times E \leqslant E \times E^*$.

Grabmayer

The following results hold for all alphabets, but for convenience we assume that $\mathcal{A}$ is finite in this section.

The *Brzozowski-derivative* E_a [12, 25, 33, 107, 108] for regular expression E and $a \in \mathcal{A}$ is defined in Figure 21.

Grabmayer [52] recognized that Brzozowski-derivatives can be combined with the ACI-properties of $+$ and the coinductive fixed point rule for recursive types of Brandt and Henglein [23] to give a *coinductive* axiomatization of regular expression equivalence. His rule COMP/FIX is given in Figure 22. Indeed, it can be seen that in the presence of a transitivity rule of equational logic, the compatibility-with-context-rules, and ACI-axioms, only the rule COMP/FIX is needed to obtain a complete system for regular expression equivalence, without the other rules of Grabmayer's

Figure 21 Definition of Brzozowski-derivative

$$
\begin{array}{rcll}
0_{a_i} & = & 0 & \\
1_{a_i} & = & 0 & \\
(a_i)_{a_i} & = & 1 & \\
(a_j)_{a_i} & = & 0 & (i \neq j) \\
(E + F)_{a_i} & = & E_{a_i} + F_{a_i} & \\
(E \times F)_{a_i} & = & E_{a_i} \times F & (o(E) = 0) \\
(E \times F)_{a_i} & = & E_{a_i} \times F + F_{a_i} & (o(E) = 1) \\
(E^*)_{a_i} & = & E_{a_i} \times E^* &
\end{array}
$$

inference system $\mathbf{cREG}_0(\Sigma)$. A sequent style presentation of COMP/FIX is as follows:

$$\frac{\Gamma, E = F \vdash_G E_a = F_a \text{ for all } a \in \mathcal{A}, \quad o(E) = o(F)}{\Gamma \vdash_G E = F}$$

Let us write $\Gamma_{\leqslant}$ and $\Gamma_{\geqslant}$ for Γ where all occurrences of $=$ in Γ are replaced by $\leqslant$, respectively $\geqslant$. We can show by rule induction that for each derivation of $\Gamma \vdash_G E = F$ there exist coercion judgements $\Gamma_{\leqslant} \vdash_{S_4} c : E \leqslant F$ and $\Gamma_{\geqslant} \vdash_{S_4} d : F \leqslant E$.

The only interesting rule to consider is COMP/FIX. By induction hypothesis, we have $\Gamma_{\leqslant}, f : E \leqslant F \vdash c_a : E_a \leqslant F_a$ and $\Gamma_{\geqslant}, g : F \leqslant E \vdash d_a : F_a \leqslant E_a$ for all $a \in \mathcal{A}$, where $o(E) = o(F)$. Note that $\models E = o(E) + \sum_{a \in \mathcal{A}} a \times E_a$. Salomaa [108] shows that $E = o(E) + \sum_{a \in \mathcal{A}} a \times E_a$ is derivable from the rules for weak equivalence (Figures 14, 15 and 16), extended with Salomaa's Axiom A_{11}: $F^* = (1 + F)^*$. (See also Grabmayer [52, Lemma 5, p. 189].) A_{11} is only required for what Frisch and Cardelli [46] call *problematic* regular expressions, regular expressions of the form G^* where $o(G) = 1$.

By applying the derivation coding of Salomaa's axiomatization from Subsection 2.3.3 to the derivation of $E = o(E) + \sum_{a \in \mathcal{A}} a \times E_a$, we know that there exist

Figure 22 Grabmayer's coinduction rule COMP/FIX

$$\frac{\begin{array}{ccc} [E = F] & & [E = F] \\ \vdots & \cdots & \vdots \\ E_{a_1} = F_{a_1} & & E_{a_n} = F_{a_n} \end{array}}{E = F} \; [o(E) = o(F)]$$

$\vdash_{S_4} c_E : o(E) + \sum_{a\in\mathcal{A}} a \times E_a \leqslant E$ and $\vdash_{S_4} d_E : E \leqslant o(E) + \sum_{a\in\mathcal{A}} a \times E_a$. This gives us the following derivable coercion judgements:

$$\Gamma_{\leqslant}, f : E \leqslant F \vdash_{S_4} d_E; (\mathrm{id}_{o(E)} + \textstyle\sum_{a\in\mathcal{A}} \mathrm{id}_a \times c_a); c_F : E \leqslant F$$
$$\Gamma_{\geqslant}, g : F \leqslant E \vdash_{S_4} d_F; (\mathrm{id}_{o(F)} + \textstyle\sum_{a\in\mathcal{A}} \mathrm{id}_a \times d_a); c_E : F \leqslant E$$

We can observe that they satisfy side condition S_1 and thus S_4. By the coinduction rule we can thus conclude:

$$\Gamma_{\leqslant} \vdash_{S_4} \mathrm{fix} f.d_E; (\mathrm{id}_{o(E)} + \textstyle\sum_{a\in\mathcal{A}} \mathrm{id}_a \times c_a); c_F : E \leqslant F$$
$$\Gamma_{\geqslant} \vdash_{S_4} \mathrm{fix} g.d_F; (\mathrm{id}_{o(F)} + \textstyle\sum_{a\in\mathcal{A}} \mathrm{id}_a \times d_a); c_E : F \leqslant E$$

This provides an alternative proof to the one based on coding Salomaa's System F_1 for concluding that $\models E \leqslant F$ implies $\vdash_{S_4} E \leqslant F$.

2.3.4 *Examples*

We continue Example 22 by implementing the function proving $(a + b)^* \leqslant a^* \times (b \times a^*)^*$ in the coercion language.

Example 35 (Denesting as coercion)**.** *The function from Example 22 can be implemented in our coercion language:*

$$\begin{aligned}\mathrm{fix} f.\quad & \mathrm{wrap}^{-1}; \mathrm{id} + \mathrm{retag} \times f; \mathrm{id} + \mathrm{distR};\\ & \mathrm{id} + (\mathrm{assoc} + \mathrm{assoc}); \mathrm{shuffle}; \mathrm{wrap} + \mathrm{id};\\ & \mathrm{id} + (\mathrm{tagR}\,; \mathrm{wrap}) \times \mathrm{id}; \mathrm{proj}^{-1} + \mathrm{id};\\ & (\mathrm{tagL}\,; \mathrm{wrap}) \times \mathrm{id} + \mathrm{id}; \mathrm{untag}\end{aligned}$$

Abbreviate $E = (a + b)^*$ *and* $F = a^* \times (b \times a^*)^*$. *We can calculate the inclusion from Example 22 as follows.*

$$
\begin{array}{rcll}
E & \leqslant & 1 + (a + b) \times E & \textit{by } \mathrm{wrap}^{-1} \\
& \leqslant & 1 + (a + b) \times F & \textit{by } \mathrm{f} \\
& \leqslant & 1 + (b + a) \times F & \textit{by } \mathrm{retag} \\
& \leqslant & 1 + ((b \times F) + (a \times F)) & \textit{by } \mathrm{distR} \\
& \leqslant & 1 + (((b \times a^*) \times (b \times a^*)^*) + (a \times F)) & \textit{by } \mathrm{assoc} \\
& \leqslant & (1 + (b \times a^*) \times (b \times a^*)^*) + ((a \times F)) & \textit{by } \mathrm{shuffle} \\
& \leqslant & (b \times a^*)^* + (a \times F) & \textit{by } \mathrm{wrap} \\
& \leqslant & (b \times a^*)^* + ((a \times a^*) \times (b \times a^*)^*) & \textit{by } \mathrm{assoc} \\
& \leqslant & (b \times a^*)^* + ((1 + a \times a^*) \times (b \times a^*)^*) & \textit{by } \mathrm{tagR} \\
& \leqslant & (b \times a^*)^* + F & \textit{by } \mathrm{wrap} \\
& \leqslant & 1 \times (b \times a^*)^* + F & \textit{by } \mathrm{proj}^{-1} \\
& \leqslant & (1 + a \times a^*) \times (b \times a^*)^* + F & \textit{by } \mathrm{tagL} \\
& \leqslant & F + F & \textit{by } \mathrm{wrap} \\
& \leqslant & F & \textit{by } \mathrm{untag}
\end{array}
$$

In the above example, the coercion is, operationally, basically the function defined in Example 22: It folds a constant-time computable function over its input list therefore runs in linear time. Kozen [76, Proposition 2.7] gives a proof of the same inclusion in his axiomatization of Kleene algebra. When encoding it as in Section 2.3.3 we obtain a similar, linear-time coercion. This raises the question whether computational interpretation of *all* proofs of the *same* containment in the axiomatizations of Salomaa, Kozen and Grabmayer yield coercions of the same, linear-time complexity. Remarkably, this is not the case, as illustrated by the next example.

Example 36 (Coercion efficiency)**.** *Consider* $a^* \times a^{**} \leqslant a^*$. *The simplest way to prove this with Kozen's rules is to start from* $a \times a^* \leqslant a^*$ *proved by* $\mathrm{tagR}\,;\mathrm{wrap}$. *By the left inference rule in Figure 20 we then get* $a^* \times a^* \leqslant a^*$. *By the right inference rule in Figure 20 we finally obtain* $a^* \times a^{**} \leqslant a^*$.

Let us consider the computational interpretation of this proof. We have two (nested) applications of the (left and right, respectively) inference rule from Figure 20. This gives quadratic *runtime. It is unclear to us whether there exists a proof using Kozen's whose computational interpretation as a functional program runs in linear time.*

It is possible to construct a linear-time coercion for $a^* \times a^{**} \leqslant a^*$ *in our coercion inference system, however. This can be systematically obtained by encoding the (minimal) proof in Grabmayer's axiomatization. In fact, the encoding of any proof in Grabmayer's axiomatization will have linear runtime. This is because the only admissible application of*

recursion in Grabmayer's axiomatization is of the form $\mathrm{fix} f.d_E; (\mathrm{id} + \Sigma_{a \in \mathcal{A}} \mathrm{id} \times c_a); c_F$*, where* f *does not occur in* d_E *and* c_F*, which entails that only constant amount of processing occurs for each constant part of the input.*

2.3.5 *Parametric completeness*

Let us extend regular expressions by adding variables that can be bound to arbitrary regular sets. Formally,

$$E, F, G, H ::= 0 \mid 1 \mid a \mid X \mid E + F \mid E \times F \mid E^*$$

where X ranges over a denumerable set of *(formal) variables* $\{X_i\}_{i \in \mathbb{N}}$. Such a regular expression is *closed* if it contains no formal variables.

We define $\models \forall X_1, \ldots, X_m. E[X_1, \ldots, X_m] \leqslant F[X_1, \ldots, X_m]$ if the containment holds for all substitutions of X_i with (closed) regular expressions.

Our axiomatization is immediately applicable to regular expressions with free variables. Without change it is not only complete, but *parametrically complete* for infinite alphabets:

Theorem 37 (Parametric completeness)**.** *Let* $\mathcal{A}$ *be infinite. Let the side condition* P *for the coinduction rule in Figure 17 be either total hereditariness or* S_2*.*
Then: $\models \forall X_1, \ldots, X_m. E \leqslant F$ *if and only if* $\vdash_P c : E \leqslant F$*.*

Proof. Only if: By rule induction, coercion axiomatization is closed under substitution, with total hereditariness or S_2 as side condition. Note that this is not the case for S_4. (Technically, S_1 is not even defined for regular expressions with variables, since $o(X)$ is undefined.)

If: Assume $\models \forall X_1, \ldots, X_n. E \leqslant F$. Let $b_1, \ldots, b_n$ be n distinct alphabet symbols not occurring in E or F. (Since $\mathcal{A}$ is infinite, they exist.) By definition of $\models \forall X_1, \ldots, X_n. E \leqslant F$, we have $\models E\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\} \leqslant F\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\}$. By Theorem 33 (completeness), there is a derivable coercion judgement $\vdash_P c : E\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\} \leqslant F\{X_1 \mapsto b_1, \ldots, X_n \mapsto b_n\}$. It can be shown that our coercion axiomatization with hereditary totality or S_2 as side condition is closed under substitution in the sense that the $b_1, \ldots, b_n$ can be replaced by arbitrary regular expressions $E_1, \ldots, E_n$, respectively, such that $\vdash c : E\{b_1 \mapsto E_1, \ldots, b_n \mapsto E_n\} \leqslant F\{b_1 \mapsto E_1, \ldots, b_n \mapsto E_n\}$. In particular, we can choose $X_1, \ldots, X_n$ for $E_1, \ldots, E_n$ and thus obtain $\vdash c : E \leqslant F$. □

As a consequence of Theorem 37, a schematic containment such as $E \times E^* \leqslant E^* \times E$ is *derivable*, not just *admissible* in our axiomatization: we can synthesize a *single* coercion judgement for it and use it for all instances of E. For finite alphabets our axiomatization is incomplete, however: $\models \forall X. (X \leqslant (a + b)^*)$ holds for $\mathcal{A} = \{a, b\}$, but $X \leqslant (a + b)^*$ is not derivable.

Figure 23 Type-directed encoding function from parse trees (values) to bit sequences

$$
\begin{array}{lcl}
\mathrm{code}(() : 1) & = & \epsilon \\
\mathrm{code}(a : a) & = & \epsilon \\
\mathrm{code}(\mathrm{inl}\, v : E + F) & = & 0\ \mathrm{code}(v : E) \\
\mathrm{code}(\mathrm{inr}\, w : E + F) & = & 1\ \mathrm{code}(w : F) \\
\mathrm{code}((v, w) : E \times F) & = & \mathrm{code}(v : E)\ \mathrm{code}(w : F) \\
\mathrm{code}(\mathrm{fold}\, v : E^*) & = & \mathrm{code}(v : 1 + E \times E^*)
\end{array}
$$

We observe that Kozen's axiomatization [76] is also parametrically complete for infinite alphabets, but not for finite alphabets. Neither Salomaa's [108] nor Grabmayer's [52] appear to be parametrically complete: In Salomaa's case we need to make a case distinction as to whether the regular expression E substituted for X has the empty word property; and in Grabmayer's case E needs to be differentiated, the proof of which depends on the syntax of E.

2.4 APPLICATION: COMPACT BIT REPRESENTATIONS OF PARSE TREES

If the regular expressions are statically known in a program we can code their elements, more precisely their parse trees, compactly as bit strings.

2.4.1 *Bit coded strings*

Intuitively, a *bit coding* of a parse tree p factors p into its static part, the regular expression E it is a member of, and its dynamic part, a bit sequence that uniquely identifies p as a particular element of E.

Consider for example the string $s = abaab$ as an element of $H_1 = (a + b)^*$. It has the unique parse tree corresponding to

$$p_s \quad = \quad [\mathrm{inl}\, a, \mathrm{inr}\, b, \mathrm{inl}\, a, \mathrm{inl}\, a, \mathrm{inr}\, b]$$

with $\vdash p_s : H_1$, which shows that $\mathrm{flat}(p_s) = abaab$ is an element of $\mathcal{L}[\![H_1]\!]$.

Figures 23 and 24 define regular-expression directed coding and decoding functions code and decode from parse trees to their *(canonical) bit codings* and back. Informally, the bit coding of a parse tree consists of listing the inl- and inr-constructors in preorder traversal, where inl is mapped to 0 and inr is mapped to 1. No bits are generated for the other constructors. For example, the bit coding b_s for p_s is 10 11 10 10 11 0.

Figure 24 Type-directed decoding function from bit sequences to parse trees (values)

$$
\begin{array}{lcl}
\mathrm{decode}'(\mathsf{d} : \mathsf{1}) & = & ((), \mathsf{d}) \\
\mathrm{decode}'(\mathsf{d} : \mathsf{a}) & = & (\mathsf{a}, \mathsf{d}) \\
\mathrm{decode}'(0\mathsf{d} : \mathsf{E} + \mathsf{E}') & = & \textbf{let}\,(v, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \textbf{in}\,(\mathrm{inl}\, v, \mathsf{d}') \\
\mathrm{decode}'(1\mathsf{d} : \mathsf{E} + \mathsf{E}') & = & \textbf{let}\,(w, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \textbf{in}\,(\mathrm{inr}\, w, \mathsf{d}') \\
\mathrm{decode}'(\mathsf{d} : \mathsf{E} \times \mathsf{E}') & = & \textbf{let}\,(v, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \quad (w, \mathsf{d}'') = \mathrm{decode}'(\mathsf{d}' : \mathsf{E}') \\
 & & \textbf{in}\,((v, w), \mathsf{d}'') \\
\mathrm{decode}'(\mathsf{d} : \mathsf{E}^*) & = & \textbf{let}\,(v, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{1} + \mathsf{E} \times \mathsf{E}^*) \\
 & & \textbf{in}\,(\mathrm{fold}\, v, \mathsf{d}') \\
\mathrm{decode}(\mathsf{d} : \mathsf{E}) = & = & \textbf{let}\,(w, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \textbf{in if}\ \mathsf{d}' = \epsilon\ \textbf{then}\, w\ \textbf{else}\, \mathit{error}
\end{array}
$$

We can think of the bit coding of a parse tree p as a bit coding of the underlying string $\mathrm{flat}(\mathsf{p})$. Note that the bit coding of a string is not unique. It depends on

- which regular expression it is parsed under; and
- if the regular expression is ambiguous, which parse tree is chosen for it.

As an illustration of the first effect, the bit representation b'_s of s under $H_2 = \mathsf{1} + (\mathsf{a} + \mathsf{b})^* \times (\mathsf{a} + \mathsf{b})$ is different from b_s: it is $1\,10\,11\,10\,10\,0\,1$. Since both H_1 and H_2 are unambiguous there are no other bit representations of s under either H_1 or H_2.

2.4.2 *Bit code coercions*

So what if we have a bit representation of a run-time string under one regular expression and we need to transform it into a bit representation under another regular expression? This arises if the branches of a conditional each return a bit-coded string, but under different regular expressions, and we need to ensure that the result of the conditional is a bit coding in a common regular expression that contains the two.

Let us consider s again and how to transform b_s into b'_s. As we have seen in Section 2.3.3, E^* is contained in $\mathsf{1} + (\mathsf{E}^* \times \mathsf{E})$ for all E and there is a parametric

polymorphic coercion $c_1 : \forall X.X^* \leqslant 1 + (X^* \times X)$ mapping any value $\vdash p : E^*$ representing a parse tree for string $s' = \mathrm{flat}(p)$ to a parse tree $\vdash p' : 1 + E^* \times E$ for s'. In particular applying c_1 to p_s yields p'_s.

We can compose c_1 with code and decode from Figures 23 and 24 to compute a function $\hat{c_1}$ operating on bit codings:

$$\hat{c_1} = \mathrm{code} \cdot c_1 \cdot \mathrm{decode}.$$

Instead of converting to and from values we can define a *bit coding coercion* by providing a computational interpretation of coercions that operates directly on bit coded strings. See Figure 25. It uses the type-directed function split from Figure 26 for splitting a bit sequence into a pair of bit sequences.

Consider for example the coercion $\vdash c_0 : E^* \times E$ to $E \times E^*$ from Section 2.3.3. By interpreting c_0 according to Figure 25 we arrive at the following highly efficient function g_E, which transforms bit codings of values of $E^* \times E$ into corresponding bit representations for $E \times E^*$.

$$\begin{array}{lcl} g_E(0d) & = & 0d \\ g_E(1d) & = & 1\, f_E(d) \\ f_E(0d) & = & d0 \\ f_E(1d) & = & \mathbf{let}\,(d_1, d_2) = \mathrm{split}_E(d)\mathbf{in}\, d_1\, 1\, f_E(d_2) \end{array}$$

The bit coded version of $c_1 : E^* \leqslant 1 + E^* \times E$ gives us a linear-time function h_E that operates directly on bit codings of (parse trees) of strings in E^* and transforms them to bit codings in $1 + E^* \times E$:

$$\begin{array}{lcl} h_E(0d) & = & 0d \\ h_E(1d) & = & 1\, g_E(d) \end{array}$$

Note that $h_{(a+b)}$ is the bit coding coercion from H_1 to H_2.
It transforms 10 11 10 10 11 0 into 1 10 11 10 10 0 1 without ever materializing a string or value.

2.4.3 *Tail-recursive μ-types*

The presented bit sequences are compact, but their sizes depend on the regular expression used. Thus it is necessary to use *reasonable* regular expressions to obtain compact bit sequences. In fact the most compact representations can be found only by generalizing regular expressions to *tail-recursive μ-types*. We will use the remainder of this section to study this extension, and the compression it allows.

Figure 25 Coercions operating on typed bit sequence representations instead of values

$$
\begin{array}{lcl}
\mathrm{retag}(0\mathrm{d}) & = & 1\mathrm{d} \\
\mathrm{retag}(1\mathrm{d}) & = & 0\mathrm{d} \\
\mathrm{retag}^{-1} & = & \mathrm{retag} \\
\mathrm{tagL}\,(\mathrm{d}) & = & 0\mathrm{d} \\
\mathrm{untag}\,(\mathrm{bd}) & = & \mathrm{d} \\
\mathrm{shuffle}(0\mathrm{d}) & = & 00\mathrm{d} \\
\mathrm{shuffle}(10\mathrm{d}) & = & 01\mathrm{d} \\
\mathrm{shuffle}(11\mathrm{d}) & = & 1\mathrm{d} \\
\mathrm{shuffle}^{-1}(00\mathrm{d}) & = & 0\mathrm{d} \\
\mathrm{shuffle}^{-1}(01\mathrm{d}) & = & 10\mathrm{d} \\
\mathrm{shuffle}^{-1}(1\mathrm{d}) & = & 11\mathrm{d} \\
\mathrm{swap}(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{swap}^{-1} & = & \mathrm{swap} \\
\mathrm{proj}(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{proj}^{-1}(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{assoc}(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{assoc}^{-1}(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{distL}(\mathrm{d} : \mathrm{E} \times (\mathrm{F} + \mathrm{G})) & & \\
 & = & \textbf{let}\,(\mathrm{d}_1, \mathrm{bd}_2) = \mathrm{split}(\mathrm{d} : \mathrm{E}) \\
 & & \textbf{in}\,\mathrm{bd}_1\,\mathrm{d}_2 \\
\mathrm{distL}^{-1}(\mathrm{bd} : (\mathrm{E} \times \mathrm{F}) + (\mathrm{E} \times \mathrm{G})) & & \\
 & = & \textbf{let}\,(\mathrm{d}_1, \mathrm{d}_2) = \mathrm{split}(\mathrm{d} : \mathrm{E}) \\
 & & \textbf{in}\,\mathrm{d}_1\,\mathrm{bd}_2 \\
\mathrm{distR}(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{distR}^{-1}(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{wrap}\,(\mathrm{d}) & = & \mathrm{d} \\
\mathrm{wrap}^{-1}(\mathrm{d}) & = & \mathrm{d} \\
(\mathrm{c} + \mathrm{c}')(0\mathrm{d}) & = & 0\ \mathrm{c}(\mathrm{d}) \\
(\mathrm{c} + \mathrm{c}')(1\mathrm{d}') & = & 1\ \mathrm{c}'(\mathrm{d}')) \\
(\mathrm{c} \times \mathrm{c}')(\mathrm{d} : \mathrm{E} \times \mathrm{F}) & = & \textbf{let}\,(\mathrm{d}_1, \mathrm{d}_2) = \mathrm{split}(\mathrm{d} : \mathrm{E}) \\
 & & \textbf{in}\,\mathrm{c}(\mathrm{d}_1)\ \mathrm{c}'(\mathrm{d}_2) \\
(\mathrm{c}; \mathrm{c}')(\mathrm{d}) & = & \mathrm{c}'(\mathrm{c}(\mathrm{d})) \\
\mathrm{id}(\mathrm{d}) & = & \mathrm{d} \\
(\mathrm{fix}\ \mathrm{f.c})(\mathrm{d}) & = & \mathrm{c}[\mathrm{fix}\ \mathrm{f.c}/\mathrm{f}](\mathrm{d})
\end{array}
$$

Figure 26 Type-directed function for splitting bit sequence into subsequences corresponding to components of product type

$$
\begin{array}{lcl}
\mathrm{split}(d : 1) & = & (\epsilon, d) \\
\mathrm{split}(d : a) & = & (\epsilon, d) \\
\mathrm{split}(0d : E + E') & = & \textbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
 & & \textbf{in}\,(0d_1, d_2) \\
\mathrm{split}(1d' : E + E') & = & \textbf{let}\,(d_1, d_2) = \mathrm{split}(d' : E') \\
 & & \textbf{in}\,(1d_1, d_2) \\
\mathrm{split}(d : E^*) & = & \mathrm{split}(d : 1 + E \times E^*) \\
\mathrm{split}(d : E \times E') & = & \textbf{let}\,(d_1, d_2) = \mathrm{split}(d : E) \\
 & & \quad (d_3, d_4) = \mathrm{split}(d_2 : E') \\
 & & \textbf{in}\,(d_1 d_3, d_4)
\end{array}
$$

A common representation of strings over an alphabet $\Sigma = \{a_1, \ldots, a_{255}\}$ of 255 characters from the Latin-1 (ISO/IEC 8859-1:1998) alphabet employs a sequence of 8-bit bytes representing each of the 255 different characters and uses the remaining byte to indicate the end of the string. This gives a total size of $8 \cdot (n + 1)$ bits to represent a string of length n.

Consider now the size of the bit sequence from Section 2.4.1 of a string under regular expression E_Σ^* where E_Σ is a sum type holding all the 255 characters in Σ. This can be written in many ways using permutations and associations of the characters. For example, if we define E_Σ as $a_1 + (a_2 + (a_3 + \ldots + a_{255}) \ldots)$ this means that the size of the bit coding of a_k is k bits long. This can be improved by ensuring that the type is balanced such that each path to a character has the same length. As there are 255 characters this means we will use 8 bits to represent each characters, leaving one path unused (so one character only uses 7 bits). Now we can look at the space required for the bit coding of a string under type E_Σ^*. Since E_Σ^* is unfolded to $1 + E_\Sigma \times E_\Sigma^*$ the representation of the empty string requires 1 bit, while the representation of other strings is 1 bit plus 8 bits for representing the first character, plus the bits to represent the rest of the string for the type E_Σ^*. Thus up to $9 \cdot n + 1$ bits are used to represent a string of length n.

Figure 27 Inhabitation rule for μ

$$\frac{v : \alpha[\mu X.\alpha/X]}{\mathrm{fold}\, v : \mu X.\alpha}$$

The reason why bit codings for regular types use one bit more per character is due to the very restrictive recursion in regular expressions. The extra bit is used to say for each character that we do not want to end the string yet. This is because we can only use the .* constructor to define recursive types, and a regular expression E^* always unfolds to $1 + E \times E^*$. Therefore we use one bit for each character to choose the right hand side after the unfold. This is equivalent to using a unary integer representation to state how many times the E inside the .* is used, and this leaves room for optimization.

We now generalize the recursion to *tail-recursive* μ*-types* in order to obtain more compact bit codings. Consider the language of expressions $\mathrm{UnReg}^{\mu}_{\Sigma}$ over a finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$:

$$\alpha ::= 0 \mid 1 \mid a \mid \alpha_1 + \alpha_2 \mid \alpha_1 \times \alpha_2 \mid \mu X.\alpha \mid X$$

We define the free variables of α to be the set of variables X that occur in α without a binder μX. If there are no free variables in α then we say that α is closed. We call α tail-recursive if α_1 is closed in all subterms of the form $\alpha_1 \times \alpha_2$.

We can now define the language $\mathrm{Reg}^{\mu}_{\Sigma}$ as the closed, tail-recursive expressions from $\mathrm{UnReg}^{\mu}_{\Sigma}$.

We need to define semantics, type-interpretation and inhabitation for the new expressions, but we can reuse the definitions from regular expressions ($\mathcal{L}[\![]\!], \mathcal{T}[\![]\!], v : E$), simply by adding new rules for the new μ and X constructs.

We can use the language semantics from Definition 10, except the definition of $\mathcal{L}[\![E^*]\!]$ must be replaced with the definitions

$$\mathcal{L}[\![X]\!] = \varnothing \text{ and } \mathcal{L}[\![\mu X.\alpha]\!] = \textstyle\bigcup_{i \geqslant 0} \mathcal{L}[\![\alpha^{(X,i)}]\!]$$
$$\text{where } \alpha^{(X,0)} = \alpha[0/X] \text{ and } \alpha^{(X,n+1)} = \alpha[\alpha^{(X,n)}/X].$$

We can extend the type-interpretation from Definition 14 with an environment σ, and replace the definition of $\mathcal{T}[\![E^*]\!]$ with

$$\mathcal{T}[\![X]\!]_\sigma = \sigma(X) \text{ and } \mathcal{T}[\![\mu X.\alpha]\!]_\sigma = \textstyle\bigcup_{i \geqslant 0} \mathcal{T}[\![\alpha]\!]_{\sigma^{(i)}} \text{ where}$$
$$\sigma^{(0)} = \sigma[X \mapsto \varnothing] \text{ and } \sigma^{(n+1)} = \sigma[X \mapsto \mathcal{T}[\![\alpha]\!]_{\sigma^{(n)}}].$$

Finally, we can use the inhabitation rules from Figure 13, except the fold rule must be replaced with the rule for μ in Figure 27.

We can now prove that $\mathrm{Reg}^{\mu}_{\Sigma}$ expresses exactly the same languages as Reg_{Σ}:

Theorem 38 (Conservativity of tail-recursive μ-types)**.**

1. *For all* $E \in \mathit{Reg}_{\Sigma}$ *there is* $\alpha \in \mathit{Reg}^{\mu}_{\Sigma}$
   such that $\{\mathrm{flat}(v) \mid \vdash v : E\} = \{\mathrm{flat}(v) \mid \vdash v : \alpha\}$.

2. *For all* $\alpha \in \mathit{Reg}^{\mu}_{\Sigma}$ *there is* $E \in \mathit{Reg}_{\Sigma}$
   such that $\{\mathrm{flat}(v) \mid \vdash v : \alpha\} = \{\mathrm{flat}(v) \mid \vdash v : E\}$.

The equivalence of regular expressions with right-regular grammars is well known [30]. Tail-recursive μ-types are like right-regular grammars, but do not *exactly* correspond to them: tail-recursive μ-types lack mutual recursion, but offer locally scoped recursion, where grammars only provide top-level recursion. [5]

Proof. (Sketch) The first statement is proved by encoding E^* as $\mu X.1 + \alpha \times X$ where α is the encoding of E. The second statement is proved by first rewriting μ-types to the form $\mu X.(\alpha \times X + \beta)$, where X is not free in α or β. Now it can be seen that $\mathcal{L}[\![\mu X.(\alpha \times X + \beta)]\!] = \mathcal{L}[\![E^* \times F]\!]$ where E is a regular expression encoding of α and F is an encoding of β. □

Even though Reg_Σ^μ expresses exactly the same languages as Reg_Σ, the new expressions allow us to define $(a + b + c)^*$ as $\mu X.(1 + a \times X) + (b \times X + c \times X)$ and thus saving us one bit per character we need to express.

Using this optimization the representation of any string with respect to the generalized regular expression type α_Σ will use eight bits per Latin-1 character plus eight bits to terminate the string. This is exactly the same size as the standard Latin1-representation. In fact the bit-representations for this type will be exactly the same as the bit-representations for the standard Latin-1 representation if the same permutation of characters is chosen in α_Σ.

It may not seem very impressive to reinvent the Latin-1 representation this way, but it can guarantee that bit coding uses at most as much space as the Latin-1 representation. The benefit of bit coding comes when we no longer consider all Latin-1 strings, but a subset specified by a regular expression. In this case the bit codings will generally be more compact. The ultimate example of this is when the regular expression allows exactly one string. For example the bit sequence of `'abcbcba'` under regular expression $a \times b \times c \times b \times c \times b \times a$ uses *zero* bits since its value contains no inl/inr-choices. Of course the program needs to know the regular expression in order to interpret the bit sequences, but that can be shared across interpretation of multiple bit sequences.

We end this section with two examples showing the bit sizes of strings under different regular expressions.

Example 39. *In the table below* Z *denotes a designated end-of-string character; and characters* a, b, c *their 8-bit Latin-1 codings.*

5 Milner [86] presents a sound and complete axiomatization of behavioral equivalence of μ-terms denoting labeled transition systems. Behavioral equivalence is properly weaker than regular expression equivalence, however. Crucially, distributivity $E \times (F + G) = E \times F + E \times G$ does not hold.

Regular expression	*Representation*	*Size*
Latin1	*abcbcbaZ*	*64*
Σ^*	*1a1b1c1b1c1b1a0*	*64*
$((a + b) + (c + d))^*$	*10010111010111010111000*	*22*
$((a + b) + c)^*$	*1001011110111101111000*	*20*
$a \times (b + c)^* \times a$	*10111011100*	*11*
$a \times b \times c \times b \times c \times b \times a$		*0*

The following is a more realistic example, where we also consider the data size before and after text compression.

Example 40 (Sizes for XML record collection string)**.** *Consider the following regular expression, corresponding to a regular XML schema ($\times$ and associativity have been omitted for simplicity).*

```
<CATALOG>
 (<CD><TITLE>Σ*</TITLE><ARTIST>Σ*</ARTIST>
     <COUNTRY>Σ*</COUNTRY><COMPANY>Σ*</COMPANY>
     <PRICE>Σ*</PRICE><YEAR>Σ*</YEAR>
 </CD>)*
</CATALOG>
```

This regular expression describes an XML-format for representing a list of CDs. We have found the sizes for representing a specific list containing 26 CDs to be the following

	Uncompressed	*Compressed*
Latin-1	*32760*	*7248*
bit representation using E_Σ	*11187*	*6654*
bit representation using α_Σ	*9947*	*6552*

As we can see, there is almost a factor 3 reduction in the space requirement when using the regular expression specific bit codings. The benefit is reduced by compression of the bit codes with bzip [109] but an 8% space reduction is still obtained.

2.5 DISCUSSION

Regular expressions are fundamental to computer science with numerous applications in semi-structured text processing, natural language processing, program analysis, graph querying, shortest path computation, compilers, program verification, bioinformatics and more.

Salomaa [108] and, independently, Aanderaa provided the first sound and complete axiomatizations of regular expression equivalence , based on a unique fixed

point rule, with Krob [80], Pratt [102] (for extended regular expressions), and Kozen [76] providing alternatives in the 90s.

Recently, coinductive axiomatizations based on finitary cases of Rutten's coinduction principle [107] for simulation relations have become popular: Grabmayer [52] for regular expressions; Chen and Pucella [29] and Kozen [78] for Kleene Algebra with Tests. Silva, Bonsangue and Rutten show how to specialize their coalgebraic framework to regular languages and show how to translate regular languages into so-called *deterministic expressions* [110, Example 3.4]. Such expressions appear to correspond to (nondeterministic) linear forms [12], which in turn represent ϵ-free nondeterministic automata. Their particular translation may generate exponentially bigger expressions than the original regular expressions, however, which may limit practical applicability.

Grabmayer's axiomatization [52] is based on Brzozowski derivatives [12, 25], which allow automata constructions and pairwise regular expression rewriting [11, 50, 83] to be understood as proof search. In this paper we present a *declarative* coinductive axiomatization of regular expression containment, generalizing Grabmayer's *algorithmic* coinductive axiomatization of regular expression equality.[6] Ours is the first axiomatization of regular expression containment with a Curry-Howard-style computational interpretation of containment proofs as functions (coercions) operating on regular expressions read as *regular types*. Like Kozen's (noncoinductive) axiomatization, but unlike Salomaa's (noncoinductive) and Grabmayer's (coinductive) it is parametrically complete for infinite alphabets: If $E[X] \leqslant F[X]$ for all X, then there is a parametric polymorphic coercion $c : \forall X.\, E[X] \leqslant F[X]$.

Frisch and Cardelli [46] were, as far we know, the first to state the precise connection between regular expressions as languages and regular expressions as types (Section 2.2.2 in this paper).

Coinductive axiomatizations with a Curry-Howard style computational interpretation have been introduced by Brandt and Henglein [23] for recursive type equivalence and subtyping, expounded on by Gapeyev et al. [47], as an alternative to the axiomatization of Amadio and Cardelli [10], which uses the unique fixed point principle. In this fashion, classical unification closure can be understood as proof search for a type isomorphism, and the product automaton construction of Kozen et al. [79] as search for the coercion embedding a subtype into another type. Di Cosmo et al. [35] provide a coinductive characterization of recursive subtyping with associative-commutative products, but it is not a proper *axiomatization* since it appeals directly to bisimilarity.[7] Recursive type isomorphisms have also been studied by Abadi and Fiore [7], Fiore [38, 39] and have been used for stub generation

6 Here, "declarative" and "algorithmic" are used in the same sense as in Benjamin Pierce's book *Types and Programming Languages*, MIT Press.

7 Bisimilarity is coinductively defined (that is, as a greatest fixed point), but not necessarily *finitarily* as required in an ordinary (recursive) axiomatization.

[15].

The (finitary) coinduction rule in its most general form is $\dfrac{\Gamma, P \vdash P}{\Gamma \vdash P}$ It requires a side condition for soundness, which is usually specific to the syntax of the formulae P and the particular logical system at hand. Numerous syntactic variations may be possible, and care must be applied to retain soundness; e.g. by not allowing a transitivity rule [47]. This work represents the end of a quest for a general *semantic* side condition, at least for containment formulae: Interpret a *proof* of containment computationally as a function, and let the side condition be that the conclusion (now with explicit proof object) under this interpretation be (hereditarily) total. For regular expression containment, hereditary totality turns out to be undecidable, but it justifies the soundness of our side conditions S_2 and S_4, which each yield sound and complete axiomatizations of regular expression containment. Their disjunction is expressive enough to facilitate a compositional encoding of derivations in Salomaa's, Kozen's and Grabmayer's axiomatizations.

Brandt and Henglein [23, Section 4.3] discuss how the finitary coinduction rule can be understood as a rule for finding a *finite* set of formulae that are intrinsically consistent. Intuitively this corresponds to constructing proofs of such formulae that are finite, but may be circular: they may contain occurrences of formulae to be proved as assumptions. This is also the essence of circular coinduction for behavioral equivalences [104]. Rosu and Lucanu [105] provide a general proof-theoretic framework for applying circular coinduction soundly. It allows marking certain equations as *frozen* to prevent them from being applied prematurely, which would lead to unsound conclusions. Our approach is fundamentally different in the following sense: We allow circular equations without any restriction. An equation is interpreted as a pair of potentially partial functions, whose definition depends on the particular (circular) proof of the equation. An equation is *valid* (sound), however, only if the two functions making up its computational interpretation are total.

Our regular-expression specific bit representation of (parse trees of) strings corresponds to composing regular expression parsing with the flattening function of Jansson and Jeuring [68], who attribute the technique to work in the 80s on text compression with syntactic source information models (see references in their paper). Bit coding captures the idea that only choices made – the tags of sum types – need to be encoded, which is also the key idea of oracle-based coding in proof-carrying code [89]. Our direct compilation of containment proofs to bit manipulation functions appears to be novel, however. It should be noted that compaction by bit coding is orthogonal to statistical text data compression. As we have illustrated, combining them may yield significantly shorter bit strings than either technique by itself.

Type- and coercion-theoretic techniques appear to be applicable to *regular expres-*

sion types [63, 64, 93, 113] and other nonregular extensions, notably context-free languages.[8] This remains to be investigated thoroughly, however. Note that regular expression types are a proper extension of regular expressions *as* types.

There are also numerous practically motivated topics to investigate: Inference of regular type containments and search for practically *efficient* coercions implementing them; disambiguation of regular expressions by annotating them; instrumenting automata constructions for fast input processing that yields parse trees; and more. We hope that this may lead the way to putting logic and computer science into a new generation of expressive, generally applicable high-performance regular expression processing tools.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comprehensive critical and constructive comments, and for detailed recommendations for improvement. Any remaining problems are solely the authors' responsibility. The alphabetically first author is grateful to Eijiro Sumii, Yasuhiko Minamide, Naoki Kobayashi, and Atsushi Igarashi for jointly solving the question of decidability of hereditary totality (Theorem 29). We would like to thank Dexter Kozen for sharing many insights on Kleene Algebras as part of a mini-course held at DIKU, as well as for ideas and suggestions for the present–and for further–work. Thanks also to Alexandra Silva for explaining and commenting her work on Kleene coalgebras and its relation to regular expressions. Finally we would like to thank the participants of our graduate course "Topics in Programming Languages: Theory and Practice of Regular Expressions", held at DIKU in Spring 2010, for exploring some of the applications of regular expressions as types.

8 Completeness is out of the question, of course, since context-free grammar containment is not recursively enumerable.

3

BIT-CODED REGULAR EXPRESSION PARSING

Authors:

Fritz Henglein - University of Copenhagen,
Lasse Nielsen - University of Copenhagen

Presented at:

LATA 2011 – 5th International Conference on Language and Automata Theory and Applications

Regular expression parsing is the problem of producing a parse tree of a string for a given regular expression. We show that a compact bit representation of a parse tree can be produced efficiently, by simplifying the algorithms of Dubé and Feeley, and Frish and Cardelli to emit the bits of the bit representation without explicitly materializing the parse tree itself. Automata minimization is important for the efficiency of the automata based algorithms. The generated transducers can be minimized using an adaptation of the algorithm by Mohri, which after a transformation of the input labels applies the DFA minimization algorithm. What happens when the DFA minimization algorithm is applied without the preceding transformation? Examples show the result is not minimal, however we prove the result to be a minimal transducer under a stronger equivalence. This provides a semantic description of the result, and suggests the result to be close to minimal. We have implemented the parsing algorithms and the transducer minimizations, gauged the parsing performance as well as the effects of the two minimizations.

3.1 INTRODUCTION

A *regular expression* over finite alphabet Σ, as introduced by Kleene [72], is a formal expression generated by the regular tree grammar

$$E, F ::= 0 \mid 1 \mid a \mid E + F \mid E \times F \mid E^*$$

where $a \in \Sigma$, and *, $\times$ and $+$ have decreasing precedence. (In concrete syntax, we may omit $\times$ and write $\mid$ instead of $+$.) Informally, we talk about a regular expression *matching* a string, but what exactly does that mean?

In classical theoretical computer science, regular expression matching is the problem of *deciding* whether a string belongs to the regular *language* denoted by a regular expression; that is, it is *membership testing* [18]. In this sense, $abdabc$ matches `((ab)(c|d)|(abc))*`, but $abdabb$ does not. This is captured in the *language interpretation* for regular expressions in Figure 28.

In *programming*, however, membership testing is rarely good enough: We do not only want a yes/no answer, we also want to obtain proper *matches* of substrings against the subexpressions of a regular expression so as to *extract* parts of the input for further processing. In a *Perl Compatible Regular Expression (PCRE)*[1] matcher, for example, matching $abdabc$ against E = `((ab)(c|d)|(abc))*` yields a substring match for each of the 4 parenthesized subexpressions ("groups"): They match abc, ab, c, and ϵ (the empty string), respectively. If we use a POSIX matcher [37] instead, we get abc, ϵ, ϵ, abc, however. The reason for the difference is that `((ab)(c|d)|(abc))*` is *ambiguous*: the string abc can match the left or the right alternative of `(ab)(c|d)|(abc)`, and returning substring matches makes this difference observable.

A limitation of Perl- and POSIX-style matching is that we only get at most one match for each group in a regular expression. This is why only abc is returned, the *last* substring of $abdabc$ matching the group `((ab)(c|d)|(abc))` in the regular expression above. Intuitively, we might expect to get the *list* of all matches $[abd, abc]$. This is possible with *regular expression types* [64]: Each group in a regular expression can be named by a variable, and the output may contain multiple matches for the same variable. For a variable under *two* Kleene stars, however, we cannot discern the matches belonging to different level-1 Kleene-star groups.

An even more refined notion of matching is thus *regular expression parsing*: Returning a parse tree of the input string under the regular expression read as a *grammar*. Automata-theoretic techniques, which exploit equivalence of regular expressions under their language interpretation, typically change the grammatical structure of matched strings and are thus not directly applicable. Only recently have linear-time[2] regular expression *parsing* algorithms been devised [36, 46].

1 See `http://www.pcre.org`.

2 This is the data complexity; that is for a fixed regular expression, whose size is considered constant.

Figure 28 The language interpreation of regular expressions

$$\begin{array}{rclcrcl} \mathcal{L}[\![0]\!] & = & \varnothing & & \mathcal{L}[\![E + F]\!] & = & \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!] \\ \mathcal{L}[\![1]\!] & = & \{\epsilon\} & & \mathcal{L}[\![E \times F]\!] & = & \mathcal{L}[\![E]\!]\,\mathcal{L}[\![F]\!] \\ \mathcal{L}[\![a]\!] & = & \{a\} & & \mathcal{L}[\![E^*]\!] & = & \{\bigcup_{i \geqslant 0} \mathcal{L}[\![E]\!]^i \end{array}$$

where ϵ is the empty string, $S\,T = \{s\,t \mid s \in S \wedge t \in T\}$, and $S^0 = \{\epsilon\}, S^{i+1} = S\,S^i$.

In this paper we show how to generate a compact *bit-coded* representation of a parse tree highly efficiently, without explicitly constructing the parse tree first. A bit coding can be thought of as an oracle directing the expansion of a grammar – here we only consider regular expressions – to a particular string. Bit codings are interesting in their own right since they are typically not only smaller than the parse tree, but also smaller than the string being parsed and can be combined with other techniques for improved text compression [27, 32].

In Section 3.2 we recall that parse trees can be identified with the elements of regular expressions interpreted as types, and in Section 3.3 we describe bit codings and conversions to and from parse trees. Section 3.4 presents our algorithms for generating bit coded parse trees. These are empirically evaluated in Section 3.5. In Section 3.6 we describe how the generated automatons can be reduced in two different ways. Both algorithms are then proved to produce a minimal transducer but under two different equivalences. Section 3.7 empirically evaluates the presented minimizations and their effects on the automaton based implementations. Section 3.8 summarizes our conclusions.

3.2 REGULAR EXPRESSIONS AS TYPES

Parse trees for regular expressions can be formalized as ad-hoc data structures [22, 36], representing exactly how the string can be expressed in the regular expression. This means that both membership testing, substring groups and regular expression types can be found by filtering away the extra information in the parse tree. Interestingly, parse trees also arise completely naturally by interpreting regular expressions as *types* [46, 56]; see Figure 29. For example, the type interpretation of regular expression $((\mathtt{ab})(\mathtt{c}|\mathtt{d})|(\mathtt{abc}))*$ is $((\{a\} \times \{b\}) \times (\{c\} + \{d\}) + \{a\} \times (\{b\} \times \{c\}))$ list. We call elements of a type *values*; e.g., $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inr}\,(a, (b, c))]$ and $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inl}\,((a, b), \mathrm{inl}\,c)]$ are different elements of $((\{a\} \times \{b\}) \times (\{c\} + \{d\}) + \{a\} \times (\{b\} \times \{c\}))$ list, and thus represent different parse trees for regular expression $((\mathtt{ab})(\mathtt{c}|\mathtt{d})|(\mathtt{abc}))*$.

We can *flatten* (unparse) any value to a string by removing the tree structure.

Figure 29 The type interpreation of regular expressions

$$\begin{array}{rclcrcl} \mathcal{T}[\![0]\!] & = & \varnothing & & \mathcal{T}[\![E+F]\!] & = & \mathcal{T}[\![E]\!]+\mathcal{T}[\![F]\!] \\ \mathcal{T}[\![1]\!] & = & \{()\} & & \mathcal{T}[\![E\times F]\!] & = & \mathcal{T}[\![E]\!]\times\mathcal{T}[\![F]\!] \\ \mathcal{T}[\![a]\!] & = & \{a\} & & \mathcal{T}[\![E^*]\!] & = & \mathcal{T}[\![E]\!]\,\mathrm{list} \end{array}$$

where $()$ is distinct from all alphabet symbols,
$S+T=\{\mathrm{inl}\,x \mid x\in S\}\cup\{\mathrm{inr}\,y \mid y\in T\}$ is the disjoint union,
$S\times T=\{(x,y) \mid x\in S, y\in T\}$ the Cartesian product of S and T, and
$S\,\mathrm{list}=\{[v_1,\ldots,v_n] \mid v_i\in S\}$ the finite lists over S.

Definition 41. *The* flattening *function* $\mathrm{flat}(.)$ *from values to strings is defined as follows:*

$$\begin{array}{rclcrcl} \mathrm{flat}(()) & = & \epsilon & & \mathrm{flat}(a) & = & a \\ \mathrm{flat}(\mathrm{inl}\,v) & = & \mathrm{flat}(v) & & \mathrm{flat}(\mathrm{inr}\,w) & = & \mathrm{flat}(w) \\ \mathrm{flat}((v,w)) & = & \mathrm{flat}(v)\,\mathrm{flat}(w) & & \mathrm{flat}(\mathrm{fold}\,v) & = & \mathrm{flat}(v) \end{array}$$

Flattening the type interpretation of a regular expression yields its language interpretation:

Theorem 42. $\mathcal{L}[\![E]\!]=\{\mathrm{flat}(v) \mid v\in\mathcal{T}[\![E]\!]\}$

A regular expression is *ambiguous* if and only if its type interpretation contains two distinct values that flatten to the same string. With p_1, p_2 as above, since $\mathrm{flat}(p_1)=\mathrm{flat}(p_2)=abdabc$, this shows that $((ab)(c|d)|(abc))^*$ is grammatically ambiguous.

3.3 BIT-CODED PARSE TREES

The description of bit coding in this section is an adaptation from Henglein and Nielsen [56]. Bit coding for more general types than the type interpretation of regular expressions is well-known [69]. It has been applied to certain context-free grammars [27], but its use in this paper for efficient regular expression parsing seems to be new.

A *bit-coded parse tree* is a bit sequence representing a parse tree for a *given* regular expression. Intuitively, bit coding factors a parse tree p into its static part, the regular expression E it is a member of, and its dynamic part, a bit sequence that uniquely identifies p as a particular element of E. The basic idea is that the bit sequence serves as an oracle for the alternatives that must be taken to expand a regular expression into a particular string.

Consider, for example, the values $p_1=[\mathrm{inl}\,((a,b),\mathrm{inr}\,d),\mathrm{inr}\,(a,(b,c))]$ and $p_2=[\mathrm{inl}\,((a,b),\mathrm{inr}\,d),\mathrm{inl}\,((a,b),\mathrm{inl}\,c)]$ from Section 3.2, which represent distinct parse

Figure 30 Type-directed encoding function from syntax trees to bit sequences

$$
\begin{array}{lcl}
\mathrm{code}((): 1) & = & \epsilon \\
\mathrm{code}(a : a) & = & \epsilon \\
\mathrm{code}(\mathrm{inl}\, v : E + F) & = & 0\ \mathrm{code}(v : E) \\
\mathrm{code}(\mathrm{inr}\, w : E + F) & = & 1\ \mathrm{code}(w : F) \\
\mathrm{code}((v, w) : E \times F) & = & \mathrm{code}(v : E)\ \mathrm{code}(w : F) \\
\mathrm{code}([v_1, \ldots, v_n] : E^*) & = & 0\ \mathrm{code}(v_1 : E) \ldots\ 0\ \mathrm{code}(v_n : E)\ 1
\end{array}
$$

trees of $abdabc$ for regular expression `((ab)(c|d)|(abc))*`. The bit coding arises from throwing away everything in the parse tree except the list and the tag constructors, which yields $[\mathrm{inl}\,\mathrm{inr}, \mathrm{inr}]$. We code inl by 0 and inr by 1, which gives us $[01, 1]$. Finally we code the list itself: Each element is prefixed by 0, and the list is terminated by 1. The resulting bit coding is $b_1 = 001\,01\,1$ (whitespace added for readability). Similarly, the bit coding of p_2 is $b_2 = 001\,000\,1$. More compact codings for lists are possible by generalizing regular expressions to tail-recursive μ-terms [56]. We stick to the given coding of lists here, however, since the focus of this paper is on constructing the bit codings, not their effect on text compression.

Figures 30 and 31 define regular-expression directed linear-time coding and decoding functions from parse trees to their bit codings and back:

Theorem 43. *If* $v \in \mathcal{T}[\![E]\!]$ *then* $\mathrm{decode}(\mathrm{code}(v : E) : E) = v$

Proof: By structural induction on v.

Note that bit codings are only meaningful in the context of a regular expression, because the same bit sequence may represent different strings for different regular expressions, and may be invalid for other regular expressions.

Bit codings are not only more compact than parse trees. As we shall see, they are also more suitable for automaton output, as it is not necessary to generate list structure, pairing or even the alphabet symbols occurring in a parse tree.

3.4 PARSING ALGORITHMS

We present two bit coding parsing algorithms in this section. The first can be understood as a simplification of Dubé and Feeley's [36] DFA-generation algorithm, producing bit codings instead of explicit parse tree. We also show that Frisch and Cardelli's [46] algorithm can be straightforwardly modified to produce bit codings.

Figure 31 Type-directed decoding function from bit sequences to syntax trees

$$
\begin{array}{lcl}
\mathrm{decode}'(\mathsf{d} : \mathsf{1}) & = & ((), \mathsf{d}) \\
\mathrm{decode}'(\mathsf{d} : \mathsf{a}) & = & (\mathsf{a}, \mathsf{d}) \\
\mathrm{decode}'(0\mathsf{d} : \mathsf{E} + \mathsf{F}) & = & \textbf{let}\,(v, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \textbf{in}\,(\mathrm{inl}\, v, \mathsf{d}') \\
\mathrm{decode}'(1\mathsf{d} : \mathsf{E} + \mathsf{F}) & = & \textbf{let}\,(w, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{F}) \\
 & & \textbf{in}\,(\mathrm{inr}\, w, \mathsf{d}') \\
\mathrm{decode}'(\mathsf{d} : \mathsf{E} \times \mathsf{F}) & = & \textbf{let}\,(v, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \quad (w, \mathsf{d}'') = \mathrm{decode}'(\mathsf{d}' : \mathsf{F}) \\
 & & \textbf{in}\,((v, w), \mathsf{d}'') \\
\mathrm{decode}'(0\mathsf{d} : \mathsf{E}^*) & = & \textbf{let}\,(v_1, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \quad (\vec{v}, \mathsf{d}'') = \mathrm{decode}'(\mathsf{d}' : \mathsf{E}^*) \\
 & & \textbf{in}\,(v_1 :: \vec{v}, \mathsf{d}') \\
\mathrm{decode}'(1\mathsf{d} : \mathsf{E}^*) & = & ([], \mathsf{d}) \\
\mathrm{decode}(\mathsf{d} : \mathsf{E}) = & = & \textbf{let}\,(w, \mathsf{d}') = \mathrm{decode}'(\mathsf{d} : \mathsf{E}) \\
 & & \textbf{in}\,\textbf{if}\,\mathsf{d}' = \epsilon\,\textbf{then}\,w\,\textbf{else}\,\mathit{error}
\end{array}
$$

3.4.1 *Dubé/Feeley-style parsing*

Our DFA-based algorithm performs the following steps: Given regular expression E and input string s,

1. generate an enhanced Thompson-style NFA with output actions (finite state transducer);
2. use the subset construction to produce an enhanced DFA with additional information on edges to capture the output actions from the NFA;
3. use the enhanced DFA as a regular DFA on s;
   - if it rejects terminate with error (no parse tree);
   - if it accepts, return the path in the DFA induced by the input string;
4. combine, in reverse order of the path, the output information on each edge traversed to construct the bit coding of a parse tree.

The steps are described in more detail below.

Enhanced NFA generation

The left column of Figure 32 shows Thompson-style NFA generation. We enhance it by adding single bit outputs to the outedges of those states that have two outgoing ϵ-transitions, shown in the right column. The output bits can be thought of indicators for an agent traversing the NFA: 0 means turn left, 1 means turn right. The other edges carry no output bit since their traversal is forced.

When traversing a path p in an enhanced NFA, the sequence of symbols read is denoted by $\mathrm{read}(p)$, and the sequence of symbols written is denoted by $\mathrm{write}(p)$.

Lemma 44 (Soundness and completeness of enhanced NFAs)**.** *Let* N_E *be the enhanced NFA for regular expression* E *according to Figure 32 (right). Then for each* $s \in \Sigma^*$ *we have*
$\{v | v \in \mathcal{T}[\![E]\!] \wedge \mathrm{flat}(v) = s\} =$
$\{\mathrm{decode}(\mathrm{write}(p) : E) \mid p$ *is a path in* N_E *from initial state to final state such that* $\mathrm{read}(p) = s\}$.

PROOF: By structural induction on E.

In other words, an enhanced NFA generates exactly the bit codings of the parse trees of the strings it accepts. Observe furthermore that no two distinct paths from initial to final state have the same output bits. This means that a bit coding uniquely determines a particular path from initial to final state, and vice versa.

Dubé and Feeley [36] also instrument Thompson-style NFAs, but with more output symbols on more edges so as to be able to generate an external representation of a parse tree. Figure 33 shows their enhanced NFA for $E = a \times (b + c)^\star \times a$ on the left. Our corresponding enhanced NFA is shown on the right.

Enhanced subset construction

During subset construction additional information is computed:

1. A map init from the NFA states q in the initial DFA state to an output $\mathrm{init}(q)$. The output $\mathrm{init}(q)$ must be $\mathrm{write}(p)$ for some path p from the initial NFA state to q where $\mathrm{read}(p) = \varepsilon$. These paths are traversed when finding the ε-closure of the initial NFA state, which is how the initial DFA state is constructed in the subset construction, and is thus easy to generate.

2. A map output_e for each edge e in the DFA, that maps each NFA state q_2 in the destination DFA state to a pair (q_1, o) of an NFA state q_1 in the source DFA state, and an output o. The output o must be $\mathrm{write}(p)$ for some path p from q_1 to q_2 where $\mathrm{read}(p)$ is the input of the DFA edge e. These paths are traversed, when the destination state of the edge is computed, and are thus simple to generate.

The DFA for the NFA from Fig. 33 (right) is shown in Fig. 34, and the result of adding the information described above is shown in Fig. 35.

The additional information captures basically the same information as in Dubé and Feeley's DFA construction, but it stores the additional information directly in the DFA edges, where Dubé and Feeley use an external 3-dimensional table. Most importantly, the additional information we need to store is reduced, since we only generate bit codings, not explicit parse trees.

Bit code construction

After accepting $s = a_1 \dots a_n$ we have a path $p = [A_0, A_1, \cdots, A_n]$ in the enhanced DFA, where $A_0, \dots, A_n$ are DFA states, each consisting of a set of NFA states, and A_0 is the initial state and A_n contains the final NFA state q_f.

We construct the bit code of a parse tree for s by calling $\text{write}(p, q_f)$ where write traverses p from right to left as follows:

$$\begin{aligned} \text{write}([A_0], q) &= \text{init}(q) \\ \text{write}([A_0, A_1, \dots, A_{k-1}, A_k], q) &= \text{write}([A_0, A_1, \dots, A_{k-1}], q') \cdot b' \\ &\quad \text{where } (q', b') = \text{output}_{A_{k-1} \to A_k}(q) \end{aligned}$$

Lemma 45 (Bit coding preservation)**.** *Let* D_E *be the enhanced DFA generated from the enhanced NFA* N_E *for* E*. If* p *is a path from the initial state in* D *to a state containing the NFA state* q *and if* $\text{write}(p, q) = b$
then there is a path p' *in* N_E *from the initial state in* N_E *to the state* q *such that* $\text{read}(p) = \text{read}(p')$ *and* $\text{write}(p, q) = \text{write}(p')$*.*

PROOF: Induction on the number of steps in p.

We can now conclude that the bit sequence found represents a parse tree for the input string.

Theorem 46 (Correctness of DFA algorithm)**.**
If D_E *is an enhanced DFA generated from an enhanced NFA* N_E *for regular expression* E*, and* p *is a path from the initial state to a final state in* D_E*, and* q_f *is the final state of* N_E *then* $\text{read}(p) = \text{flat}(\text{decode}(\text{write}(p, q_f) : E))$*.*

PROOF: This follows from first applying Lemma 45 and then Lemma 44.

Once the DFA has been generated, this method results in very efficient regular expression parsing. The DFA traversal takes time $\Theta(|s|)$, and the bit code generation takes time $\Theta(|s| + |b|)$, where $|b|$ is the length of the output bit sequence. This means the total run time complexity of parsing is $\Theta(|s| + |b|)$ which is (sequentially) optimal, since the entire string must be read, and the entire bit sequence must be written.

Example. Consider the enhanced DFA for the regular expression $a(b+c)^\star a$ in Fig. 35. If we use it to accept the string $abcba$, then we get the path $p = 0 \to 1 \to 3 \to 4 \to 3 \to 2$. Tracing the path backwards, keeping track of the output bit-sequences b and NFA states q we get the steps in the table on the right. Since $\text{init}_0 = ""$ we get the bit code $b = "" \cdot "00" \cdot "01" \cdot "00" \cdot "1" \cdot "" = "0001001"$.

DFA steps				
Step	q	Symbol	new q	b
$3 \to 2$	9	a	8	""
$4 \to 3$	8	b	3	"1"
$3 \to 4$	3	c	4	"00"
$1 \to 3$	4	b	3	"01"
$0 \to 1$	3	a	0	"00"

We can verify that the DFA parsing algorithm has given us a correct bit coding since $\text{flat}(\text{decode}("0001001" : a(b+c)^\star a)) = abcba$.

3.4.2 *Frisch/Cardelli-style parsing*

Instead of building a Thompson-style NFA from the regular expressions, Frisch and Cardelli [46] build an NFA with one node for each position in the regular expression, with the final state as the only additional node. The regular expression positions are identified by the path used to reach the position. The positions $\lambda_{\mathsf{end}}(E)$ in a regular expression E are defined as lists of choices (the choices are **fst** and **snd** for sequence, **lft** and **rgt** for sum and **star** for $\star$). $E.l$ is used to denote the subexpression of E found by following the path l. The transitions $\delta(E)$ in the NFA of a regular expression E are defined using a successor relation **succ** on the paths.

The NFA is used to generate a table $Q(l, i)$, which maps each position $l \in \lambda_{\mathsf{end}}(E)$ and input string position i to `true`, if $E.l$ accepts the ith suffix of s and `false` otherwise.

The table Q can be constructed by starting with $Q(l, i) = \texttt{false}$ for all $l \in \lambda_{\mathsf{end}}(E)$ and $i = 1 \ldots |s|$, and calling **SetPrefix**$(Q, \mathsf{end}, |s|)$, which updates Q as defined below.

SetPrefix$(Q, l, i) =$ if $Q(l, i)$ then return;
$Q(l, i) := \texttt{true}$;
for $(l', \varepsilon, l) \in \delta(E)$ do **SetPrefix**(Q, l', i);
if $i > 0$ then for $(l', s[i], l) \in \delta(E)$ do **SetPrefix**$(Q, l', i-1)$;

The run time cost and memory consumption of computing Q is asymptotically bounded by the size of Q, which is in $\Theta(|s| \cdot |E|)$.

After Q is built, it is easy to check whether s matches the regular expression E, simply by looking up $Q([], 0)$. We modify Frisch and Cardelli's build function to construct a bit coding representing the greedy leftmost (or first and greedy [117]) parse tree for s, if s matches E, as follows (notice that $l :: x$ means appending x after l):

$$
\begin{aligned}
\textbf{build}(l, i) = \quad & \text{case } E.l \text{ of} \\
& a\text{: return } (\varepsilon, i+1) \\
& 1\text{: return } (\varepsilon, i) \\
& E_1 \times E_2\text{: let } (b_1, j) = \textbf{build}(l :: \textbf{fst}, i) \\
& \qquad \text{in let } (b_2, k) = \textbf{build}(l :: \textbf{snd}, j) \text{ in return } (b_1 b_2, k); \\
& E_1 + E_2\text{: if } Q(l :: \textbf{lft}, i) \\
& \qquad \text{then let } (b_1, j) = \textbf{build}(l :: \textbf{fst}, i) \text{ in retuen } (0 b_1, j) \\
& \qquad \text{else let } (b_2, j) = \textbf{build}(l :: \textbf{snd}, i) \text{ in return } (1 b_2, j); \\
& E_1^\star\text{: if } Q(l :: \textbf{star}, i) \\
& \quad \text{then let } (b_1, j) = \textbf{build}(l :: \textbf{star}, i) \\
& \qquad \text{in let } (b_2, k) = \textbf{build}(l, j) \text{ in return } (0 b 1 b 2, k) \\
& \quad \text{else return } (1, j);
\end{aligned}
$$

The run time cost and memory consumption of $\textbf{build}([], 0)$ is asymptotically bounded by the number of cells in Q, which is in $\Theta(|E| \cdot |s|)$ and which is therefore the time complexity and memory consumption of the entire algorithm.

Figure 32 NFA generation schema

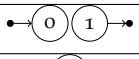
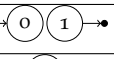
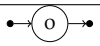
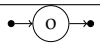
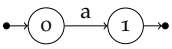
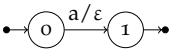
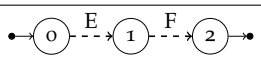
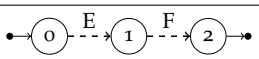
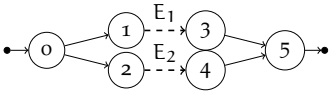
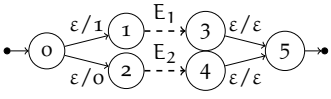
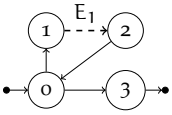
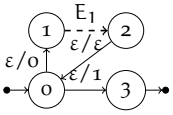
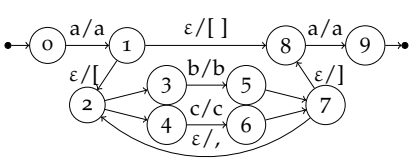
E	NFA	Enhanced NFA
0		
1		
a		
$E_1 \times E_2$		
$E + F$		
$E_1^\star$		

Figure 33 Enhanced NFAs for $a(b + c)^\star a$



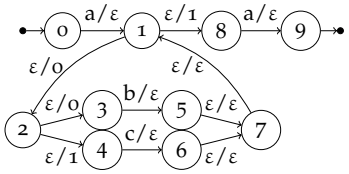
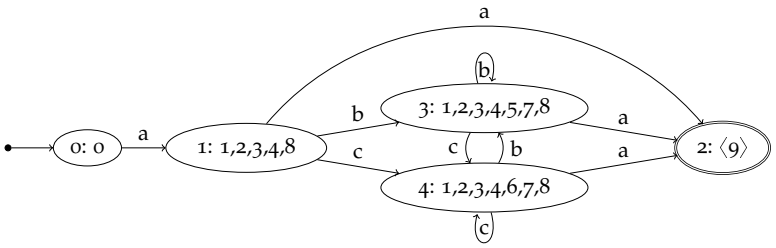


Figure 34 DFA for $a(b+c)^\star a$

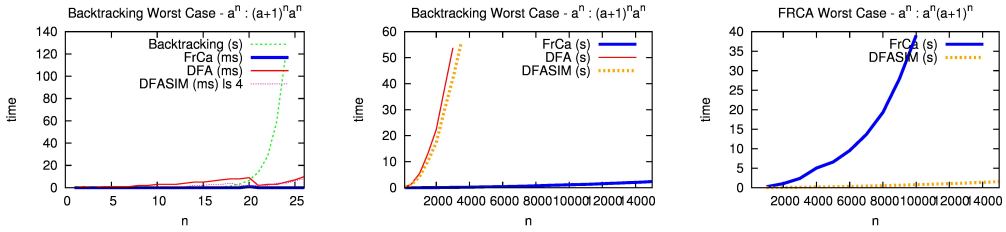


3.5 EMPIRICAL EVALUATION OF ALGORITHMS

We have implemented the algorithms described in Section 3.4 as a C++ library [92] and performed a series of performance tests on a PC with a 2.50GHz Intel Core2 Duo CPU and 4Gb of memory, running Ubuntu 10.4. We test four different parsing methods. NFA based backtracking (backtracking), implemented by a depth-first search for an accepting path in our enhanced Thompson-style NFA. FRCA is the algorithm based on Frisch and Cardelli from Section 3.4.2. DFA is the algorithm based on Dubé and Feeley from Section 3.4.1. DFASIM is the same algorithm as DFA, but where the nodes and edges of the DFA are not precomputed, but generated dynamically by need.

3.5.1 *Backtracking worst case: ($a^n : (a+1)^n a^n$)*

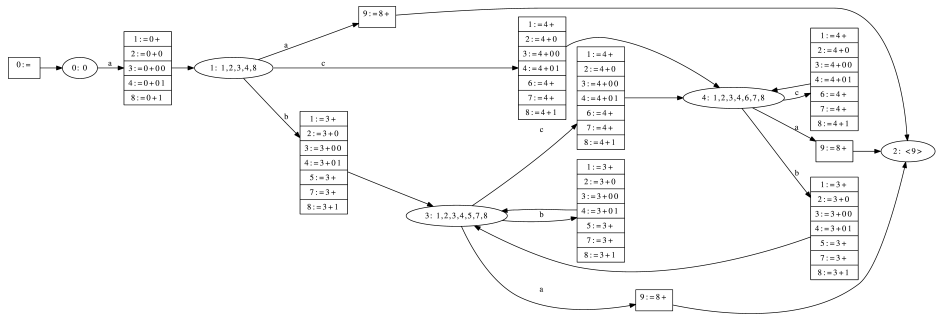
The regular expression is $(a+1)^n a^n$, where we use the notation E^n to represent $E \times \cdots \times E$ (n copies). This is a well-known example [34], which captures the problematic cases for backtracking.[3] The results of matching a^n (denoting n as) to $(a+1)^n \times a^n$ are in the two leftmost graphs below.



When parsing a string with n as, the backtracking algorithm traverses 2^n different paths before eventually finding the match. The cost of generating the DFA is $\Theta(n^2)$ ($2 \cdot n$ nodes containing n NFA-nodes on average). Since only half of the DFA-nodes are used, DFASIM is faster than generating the whole DFA, but the run time

3 If a fixed regular expression is preferred, then $(a+a)^\star \times b$ or $(a^\star \times a)^\star \times b$ provokes the same behavior.

Figure 35 Enhanced DFA for $a(b+c)^\star a$

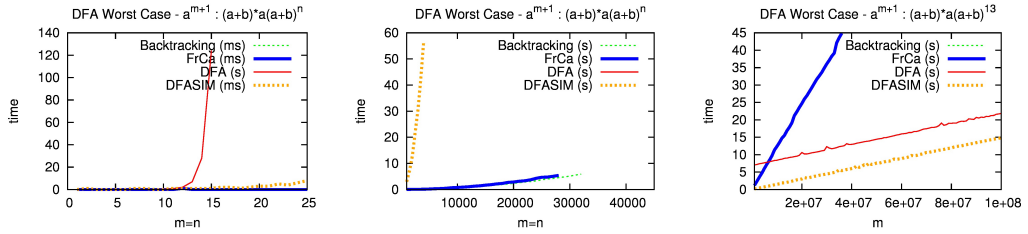


complexity is still $\Theta(n^2)$. The time used for FRCA is $\Theta(n)$, since there is exactly one suffix that can be parsed from each position in the regular expression. The reason FRCA performs much better than the other algorithms in this example is that the example was designed to be hard to parse from the left to right, and FRCA processes the string in its first phase from right to left. If we change the regular expression to $a^n(1+a)^n$, it becomes hard to process from right to left, as shown in the rightmost graph above. It is generally advantageous to process a string in the "more deterministic" direction of the regular expression.

3.5.2 *DFA worst case ($a^{m+1} : (a+b)^\star a(a+b)^n$)*

The following is a worst-case scenario for the DFA based algorithm, and a best-case scenario for the FRCA and backtracking algorithms. The regular expression is $(a+b)^\star a(a+b)^n$, and the string is a^{m+1}.

The two leftmost graphs below show the execution time when $n = m$, and the right graph shows the execution time when $n = 13$. When n is fixed to 13, the runtime of both FRCA, DFASIM and DFA are linear, but even though DFA has a large initialization time for building the DFA, FRCA uses more time for large m, because it uses more time per character.



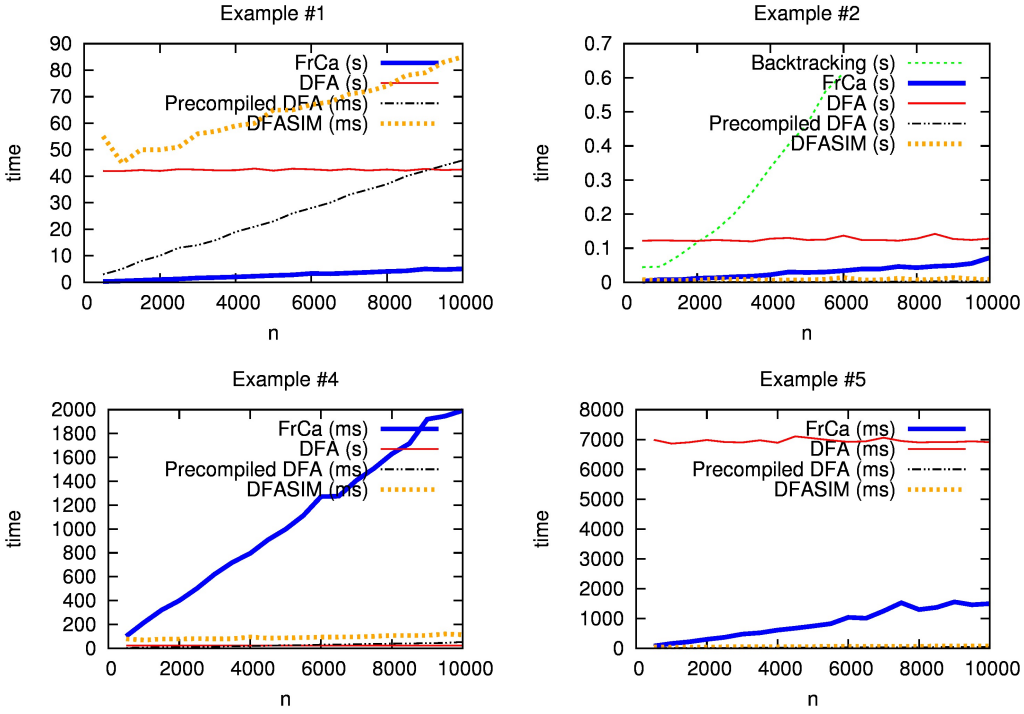
The DFA will have $\mathcal{O}(2^n)$ DFA-nodes. This causes the DFA algorithm to have a run time complexity of $\Theta(m + n \cdot 2^n)$. This exponential behaviour is avoided by the

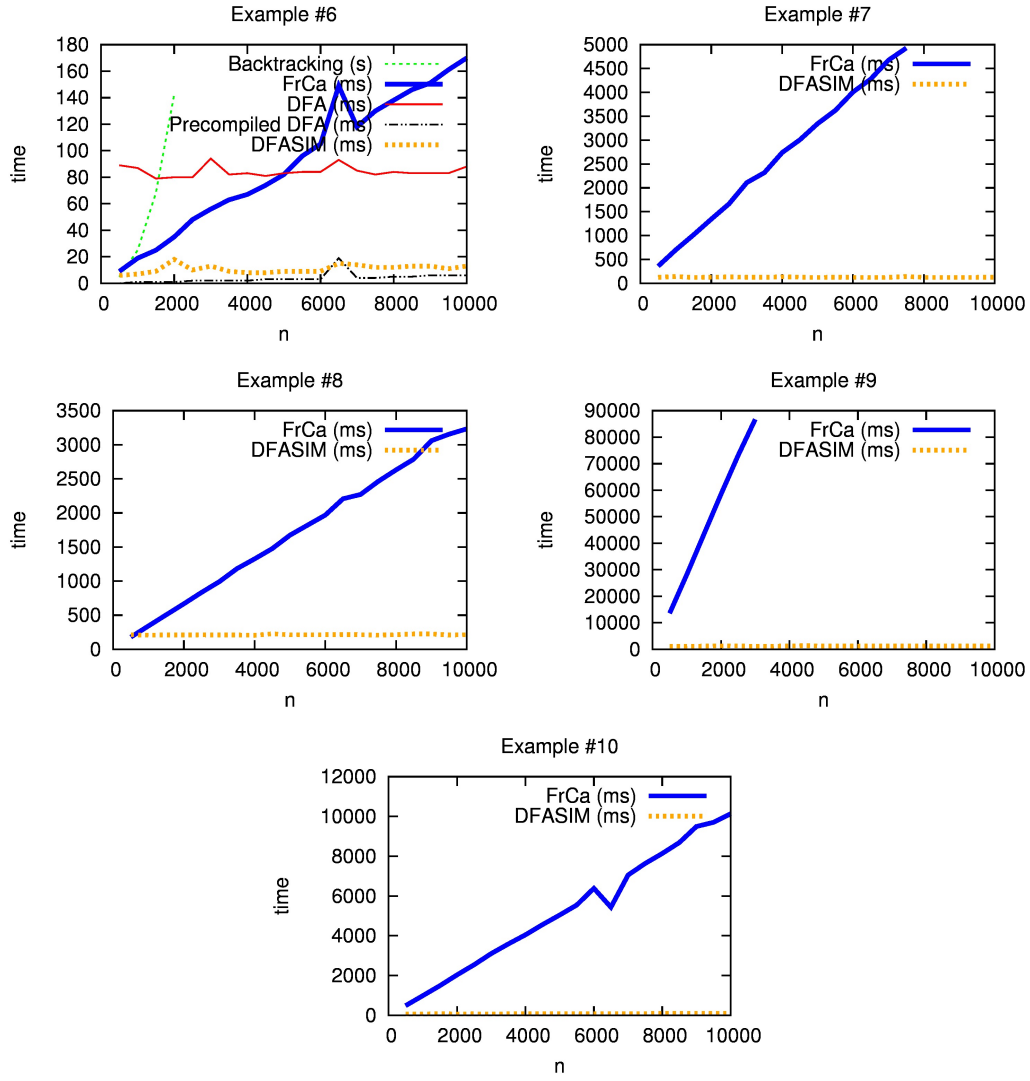
DFASIM algorithm, which only builds as many states as needed for the particular input string, which results in a $\Theta(m \cdot n)$ run time.

3.5.3 *Practical examples*

We have tested 9 of the 10 examples of "real world" regular expressions from Veanes et al. [118] to compare the performance of each algorithm. (Their example nr. 3 is uninteresting for performance testing since it only accepts strings of a bounded length).

Examples 1,4,5,7,8,9,10 are different ways of expressing the language of email addresses, while Example 2 defines the language of dollar-amounts, and Example 6 defines the language of floating point values.





The DFA and Precompiled DFA (a staged version of DFA) graphs are missing in the four last examples. This is because the DFA generation runs out of memory. The two best algorithms for these tests are FRCA and DFASIM, with DFASIM being faster by a large factor (at least 10) in all cases. Apart from the direction of processing NFA-nodes in their respective first passes, the key difference between DFASIM and FRCA is that DFASIM memoizes and reuses DFA-states at multiple positions in the input string, whereas FRCA essentially produces what amounts to a separate DFA-state for each position in the input string. In comparison to FRCA, the DFA-state memoization not only saves space, but also computation time whenever the same transition is traversed more than once.

3.6 TRANSDUCER REDUCTION

Just like for NFAs and DFAs, it is advantageous to minimize the enhanced automatons for space and time optimization. The usual approach is to generate the DFA and then apply a minimization algorithm to obtain the DFA with least possible nodes. The process of first generating the large DFA before minimization is time and space consuming, and in some cases even unfeasible because the unreduced DFA is so large that it cannot be represented in memory, even though the reduced DFA can. This approach cannot be applied to the enhanced DFAs, because the DFA minimization algorithm works by finding equivalent states which are then merged, and the merging of equivalent states is not possible when the DFA edges have output maps, because the edges that are merged may have different maps, and there is no known way to safely merge the different maps. Therefore we focus on the reduction of the generated transducers, because a transducer with less redundancy will result in a smaller enhanced DFA, since the states that differ only in the redundant transducer nodes will not be separated, and because this reduction is performed before the enhanced DFA generation, the unreduced enhanced DFA never has to be represented. Transducer minimization is undecidable in general, but can be solved in special cases. Mohri [87] created an algorithm to minimize sequential transducers, and Breslauer [24] used suffix trees to improve the runtime of the algorithm. A transducer is sequential when its projection as an input automaton is an ε-free DFA. The transducers we generate are similar, because their projection as output-automatons are DFAs but not ε-free. We adapt the algorithm by Mohri to minimize the generated transducers. The difference between being input and output deterministic is unimportant as the input and output can be switched before and after minimization, equivalently the use of input and output labels can be switched inside the algorithm, and this is what we have done. The primary modification of the algorithm is thus the elimination of ε-output edges prior to using the original minimization algorithm. The original minimization algorithm has two steps, where the first is a transformation of the input edges followed by an adaptation of the DFA minimization algorithm [8, p.141-144] to output-deterministic transducers. What happens when the first step is omitted, and the adapted DFA minimization algorithm is applied directly? Examples show that the result is not minimal, but we prove that the result can be characterized as a minimal equivalent transducer under a stronger equivalence.

3.6.1 *Transducer semantics*

We start by giving the standard definitions of transducers, paths and some definitions of path traversals. This is used to define two different semantics for transducers.

Definition 47 (Transducer)**.**
A transducer T *is a tuple* $\mathsf{T} = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ *where* Q *is the finite set of states,* Σ_i *is the input alphabet,* Σ_o *is the output alphabet,* $q_i \in Q$ *is the initial (start) state,* $Q_f \subseteq Q$ *is the set of final states and* $t \subseteq Q \times (\Sigma_i\ \mathtt{list}) \times (\Sigma_o \uplus \{\varepsilon\}) \times Q$ *is the transition relation.*

We assume that the input and output alphabets are non overlapping, that is $\Sigma_i \cap \Sigma_o = \varnothing$*.*

Definition 48 (Path and traversals)**.**
If $\mathsf{T} = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ *then a path* p *is a list of edges from* t *where the endnode for* $p[i]$ *is the startnode for* $p[i+1]$ *for* $i = 1, 2, \ldots, |p| - 1$*. If the startnode of* $p[1]$ *is* q_1 *and the endnode of* $p[|p|]$ *is* q_2 *then we say that* p *is a path in* T *from* q_1 *to* q_2*. If* p *is a path in* T *we define* $tr_i(p)$ *in* $\Sigma_i\ \mathtt{list}$ *as the input read when traversing* p *by*

$$\begin{array}{rcl} tr_i([]) & = & [] \\ tr_i((\_, is, \_, \_) :: p') & = & is@tr_i(p') \end{array}$$

If p *is a path in* T *we define* $tr_o(p)$ *in* $\Sigma_o\ \mathtt{list}$ *as the output written when traversing* p *by*

$$\begin{array}{rcl} tr_o([]) & = & [] \\ tr_o((\_, \_, \varepsilon, \_) :: p') & = & tr_o(p') \\ tr_o((\_, \_, o, \_) :: p') & = & o :: tr_o(p') \end{array}$$

If p *is a path in* T *we define* $tr_{oi}(p)$ *in* $\Sigma_i \uplus \Sigma_o\ \mathtt{list}$ *as the merging of the output written and the input read when traversing* p *by*

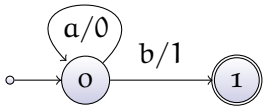
$$\begin{array}{rcl} tr_{oi}([]) & = & [] \\ tr_{oi}((\_, is, o, \_) :: p') & = & o :: is@tr_o(p') \\ tr_{oi}((\_, is, \varepsilon, \_) :: p') & = & is@tr_o(p') \end{array}$$

Transducers can be used to translate. Given an input string, a path accepting the input string is found, and the output produced when traversing that path is the result of the translation. For this reason, the standard semantics for transducers is the input to output relation that is obtained by the described procedure.

Definition 49 (Semantics)**.**
We define $\mathcal{L}_t[\![\mathsf{T}]\!]$ *as the translation relation for the transducer* T*. We define* $\mathcal{L}_{oi}[\![\mathsf{T}]\!]$ *as the output-input language accepted by the transducer* T*, which is obtained by merging the outputs and inputs traversed by the accepting paths. The formal definitions are given below*

$$\begin{array}{rcl} \mathcal{L}_t[\![\mathsf{T}]\!] & = & \{(tr_i(p), tr_o(p)) \mid p \textit{ is a path from } q_i \textit{ to some } q_f \in Q_f\}, \\ \mathcal{L}_{oi}[\![\mathsf{T}]\!] & = & \{tr_{oi}(p) \mid p \textit{ is a path from } q_i \textit{ to some } q_f \in Q_f\}. \end{array}$$

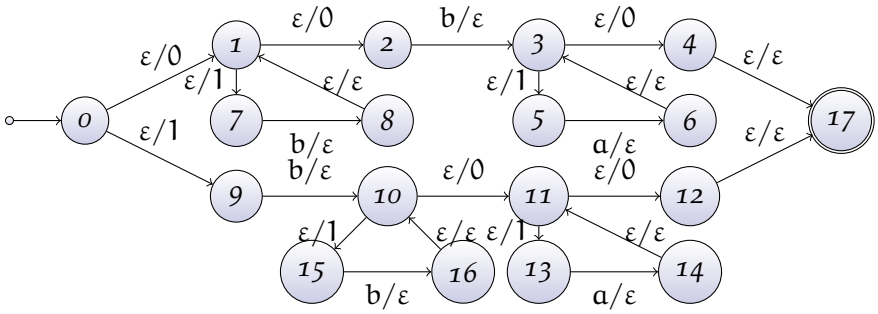
Although transducers are formalised as above, we will continue to use graphs to describe transducers in an intuitive way. We use the notation for illustrating input-automatons as graphs, and illustrate the output of transitions on the edge, separated from the input by a slash (/).
For example, the transducer $(\{0,1\},\{a,b\},\{0,1\},0,\{1\},\{(0,[a],0,0),(0,[b],1,1)\})$ can be illustrated as below.



In stead of the list representation of inputs, we simply write the input string, and we use the ε symbol for the empty input.

We will now generate an example transducer to explain the properties that we will be relying on in the generated transducers.

Example 50. *Consider the regular expression* $b^*ba^*|bb^*a^*$. *By applying the method from Section 3.4.1, the following transducer is obtained.*



The found transducer is deterministic in the output. That is, each node with more that one outgoing edge, has different output symbols on each edge and there are no edges from the final node. This means that for each sequence of output symbols, there is at most one path from the initial node to a final node which produces the given sequence of outputs. This is true for all transducers generated in this way.

Definition 51 (Output-deterministic)**.**
A transducer $T = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ *is output-deterministic if the following two properties are fulfilled.*

- *If* (q_1, is_1, x, q_2) *and* (q_1, is_2, y, q_3) *are two different edges in* t *then* $x \neq \varepsilon$ *and* $y \neq \varepsilon$ *and* $x \neq y$.
- *If* (q_1, is, o, q_2) *is in* t *then* $q_1 \notin Q_f$.

The generated transducers are always output-deterministic because the fragments from Figure 32 used to compose the transducers are output deterministic, and therefore they can be reduced using a method similar to DFA-minimization, as we will describe in the following section. Before describing the reduction algorithm, we will define equivalence relations based on the given semantics, which are used to describe the reduced transducers semantically.

Definition 52. *If* T *and* T' *are transducers then we write* $T =_t T'$ *iff* $\mathcal{L}_t[\![T]\!] = \mathcal{L}_t[\![T']\!]$.

This means that $=_t$ is an equality based on the translations of the transducers.

Definition 53. *If* T *and* T' *are transducers then we write* $T =_{oi} T'$ *iff* $\mathcal{L}_{oi}[\![T]\!] = \mathcal{L}_{oi}[\![T']\!]$.

Since the input and output languages are disjunct, each element in $\mathcal{L}_{oi}[\![T]\!]$ is canonically mapped to elements in $\mathcal{L}_t[\![T]\!]$, and thus $\mathcal{L}_t[\![T]\!]$ can be found from $\mathcal{L}_{oi}[\![T]\!]$. Therefore if $T =_{oi} T'$ then $T =_t T'$, which means that $=_{oi}$ ensures that the translation-semantics is preserved.

3.6.2 *Reduction algorithm*

The reduction algorithm presented below is an adaptation of Mohri's minimization algorithm for sequential transducers to output deterministic transducers. The algorithm uses the following steps

1 Remove states that are dead or unreachable

2 Make compact (eliminate ε-output edges)

3 Make prefix form (transform input labels)

4.a Divide the nodes into two (unmarked) groups: Q_f and $Q \backslash Q_f$

4.b Pick any unmarked group and call it G. If G is consistent, then it is marked, otherwise G is split into its maximal consistent sub-groups, replacing G and all groups are unmarked.

5 If all groups are marked, all nodes that are in the same group are equivalent, and are merged to a single node. Duplicate edges are removed.

The steps 4.a, 4.b and 5 corresponds to the well known DFA-minimization algorithm [8, p.141-144].

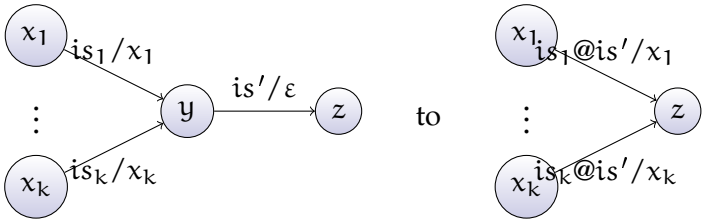
Step 2 relies on the notion of compact transducers. We will now define what we mean by a compact transducer, and describe how such a transducer is obtained.

Definition 54 (Compact Transducer)**.** *A transducer* $T = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ *is compact if it has no* ε/ε *edges, only the initial node can have outgoing edges with* ε*-output, and if it does it has no incoming edges. That is if* $(q_1, is, \varepsilon, q_2) \in t$ *then* $is \neq []$, $q_1 = q_i$ *and* $\forall (q_1', is', x', q_2') \in t . q_2' \neq q_i$.

The first condition is ensured by removing ε/ε edges by rewriting

and removing the node y by redirecting all edges with destination y to z. If y is the initial node q_i then q_i is reassigned to z. y cannot have multiple outgoing edges, as the transducer is assumed to be output-deterministic, and the rewriting does not break output-determinism or change the $\mathcal{L}_{oi}[\![T]\!]$ semantics.

The second condition is ensured by rewriting subgraphs of the form



and removing the node y when y is not the initial node q_i. Should y be the initial node q_i then the edges from the x-nodes are altered, but the y-node and the edge from the y-node are preserved. Again y cannot have multiple outgoing edges, as the transducer is assumed to be output-deterministic. This way we can ensure that only the initial node can have outgoing edges with ε-output, and if this is the case, the initial node has no incoming edges. This rewriting does not break output-determinism, or change the $\mathcal{L}_{oi}[\![T]\!]$ semantics. In case the initial node has no input-prefix – as assumed by Mohri's algorithm [87] – even the initial node will have no ε-output edges, and the resulting transducer will be entirely free of ε-output edges.

Step 3 uses the prefix form, which requires each node in the transducer to have the empty input-prefix, as defined below.

Definition 55 (Input-prefix). *Let* $T = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ *be any transducer, and* $q \in Q$ *be a live node in* T. *The input-prefix of* q *in* T *is then the longest common prefix of the set of strings* $\{tr_i(p) \mid p$ *is a path in* T *from* q *to a final node* $f \in Q_f\}$.

The input-prefix for each node in a transducer can be found by iteratively computing

$$pre(q) = \begin{cases} \varepsilon & \text{if } q \in Q_f \\ \text{largest common prefix of } \{is@pre(q') \mid (q, is, o, q') \in t\} & \text{otherwise} \end{cases}$$

until a fixpoint is reached. A more efficient algorithm is described in [24]. When the input-prefixes are found for each node, the prefix form of the transducer can be constructed by applying the transformation

$$(q, is, x, q') \to (q, pre(q)^{-1}(is@pre(q')), x, q')$$

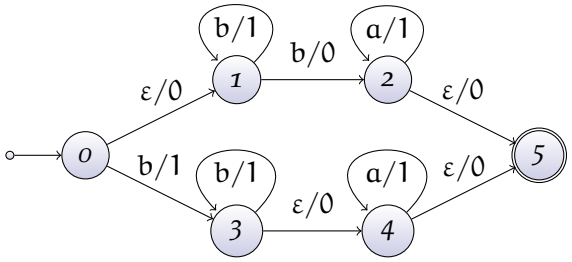
, where $s^{-1}(st) = t$, to each transition in the transducer. This results in the prefix form of the transducer, assuming the initial node has the empty prefix. In case the initial node does not have the empty prefix, a new initial node i' has to be injected along with the transition $(i', pre(i), \varepsilon, i)$. In this case we cannot be sure the minimization produces a minimal transducer, but a transducer with at most one more state than a minimal transducer.

Step 4.b uses the notion of consistent groups. To check if two nodes are consistent, we check that for each output symbol (and ε) the unique edges from the nodes with the given output symbol has the destination in the same group, and reads the same input. We say a group is consistent if all the nodes in the group are pairwise consistent. This is easily checked by building a table with one row for each node in the group, one column for each output (including ε), where each cell holds the group reached and the input read by following the edge from the node. The group is consistent if all rows are identical, and otherwise the maximal consistent sub-groups are found by grouping all the nodes with identical rows.

Example 56. *To explain the details of the described algorithm, we will apply it to the transducer from Example 50.*

The first step is to eliminate all dead and unreachable states. This does not change the transducer from Example 50, as it has no dead or unreachable states.

The second step is to make the transducer compact. Applying the described compactification method to the transducer from Example 50 results in the following compact transducer.



The third step is to produce the prefix form of the transducer, but we will skip this step to study the consequences.

The first part of the fourth step (4.a) is to create the set $Q \backslash Q_f = \{0, 1, 2, 3, 4\}$ *containing the non-final nodes, and the set* $Q_f = \{5\}$ *containing the final nodes.*

The fourth step is to check the consistency of the sets and split or mark the sets, until all sets have been marked as consistent. This in done below.

$G_{0,1,2,3,4}$	o	1	ε
o	$(G_{0,1,2,3,4},\varepsilon)$	$(G_{0,1,2,3,4},b)$	-
1	$(G_{0,1,2,3,4},b)$	$(G_{0,1,2,3,4},b)$	-
2	(G_5,ε)	$(G_{0,1,2,3,4},a)$	-
3	$(G_{0,1,2,3,4},\varepsilon)$	$(G_{0,1,2,3,4},b)$	-
4	(G_5,ε)	$(G_{0,1,2,3,4},a)$	-

$\Rightarrow$ *The group is split into its maximal consistent subgroups:* $G_{0,3}$, G_1 *and* $G_{2,4}$

$G_{0,3}$	o	1	ε
o	(G_1,ε)	$(G_{0,3},b)$	-
3	$(G_{2,4},\varepsilon)$	$(G_{0,3},b)$	-

$\Rightarrow$ G_0, G_3

$G_{2,4}$	o	1	ε	
2	(G_5,ε)	$(G_{2,4},a)$	-	$\surd$
4	(G_5,ε)	$(G_{2,4},a)$	-	

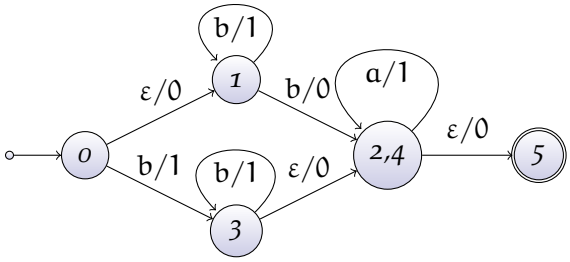
G_0	o	1	ε	
o	(G_1,ε)	(G_3,b)	-	$\surd$

G_1	o	1	ε	
1	$(G_{2,4},b)$	(G_1,b)	-	$\surd$

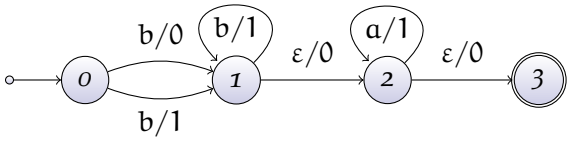
G_3	o	1	ε	
3	$(G_{2,4},\varepsilon)$	(G_3,b)	-	$\surd$

G_5	o	1	ε	
5	-	-	-	$\surd$

The result shows that nodes 2 and 4 can be merged to a single node, which results in the following transducer.



After all this work, we can take joy in the fact, that we have found a much smaller transducer which expresses the same translation as the original one. However, we have not found a minimal transducer, as the following is a smaller transducer, representing the same translation.



In order to obtain a minimal transducer, we have to realize that the input b on the edge from node 1 to node 2 can be moved to the edge from node 0 to node 1 without changing the $\mathcal{L}_t[\![T]\!]$ semantics, and this is exactly what the skipped step 3 does. After this alteration the nodes 1 and 3 are equivalent and can be merged by

step 4 and 5. It is this kind of reductions that are not found when skipping step 3. Even though moving the input preserves the $\mathcal{L}_t[\![T]\!]$ semantics, it does change the $\mathcal{L}_{oi}[\![T]\!]$ semantics, because the order between the inputs and outputs are changed.

3.6.3 *Reduction correctness*

Since step 2 results in an equivalent output-deterministic and ε-output free transducer, the theorem from [87] concludes that the described algorithm returns a minimal transducer that is $=_t$-equivalent to the input transducer.

We will now prove that although the algorithm does not return a minimal transducer that is equivalent under $=_t$ when step 3 is skipped, it does return a minimal transducer that is $=_{oi}$-equivalent to the input transducer.

The semantic states for a transducer T is the set of suffix-languages of $\mathcal{L}_{oi}[\![T]\!]$ starting with an output symbol, where the empty language is excluded, but the full language is included (even if it does not start with an output symbol or it is the empty language). The definitions for suffix-language and semantic states are given below.

Definition 57 ($L \downarrow s$)**.** *For any set of strings* L *and any string* s *we define*

$$L \downarrow s = \{s' \mid s@s' \in L\}$$

Definition 58 ($\mathcal{S}[\![T]\!]$)**.** *For any transducer* $T = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ *we define*

$$\mathcal{S}[\![T]\!] = \{\mathcal{L}_{oi}[\![T]\!] \downarrow s \mid \exists o \in \Sigma_o . s \in \mathcal{L}_{oi}[\![T]\!] \ \vee \ s@(o :: s') \in \mathcal{L}_{oi}[\![T]\!]\} \backslash \{\{\}\} \cup \mathcal{L}_{oi}[\![T]\!]$$

The semantic states have a special meaning for output-deterministic transducers, as it is necessary for them to have at least one node for each semantic state. This means that the number of nodes in an output-deterministic transducer has the number of semantic states for its language as a lower bound.

Lemma 59 (Transducer states lower bound)**.** *If* $T = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ *is an output-deterministic transducer, then* $|Q| \geqslant |\mathcal{S}[\![T]\!]|$.

Proof: By generation of an injective function from $\mathcal{S}[\![T]\!]$ to Q. See Appendix B.1.1 for details.

Assume $T^c = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$ is a compact transducer, with $q_1, q_2 \in Q$. We write $q_1 \sim q_2$ if q_1 and q_2 are in the same group after step 4. For all $q \in Q$ we consider $T^c_q = (Q, \Sigma_i, \Sigma_o, q, q_f, t)$. We can now prove that the merging of nodes in step 5 is sound and complete in the sense that only nodes with identical languages are merged, and if nodes have identical languages then they are merged.

Lemma 60 (Soundness of $\sim$)**.** *If* $q_1 \sim q_2$ *and* $s \in \mathcal{L}_{oi}[\![T_{q_1}]\!]$ *then* $s \in \mathcal{L}_{oi}[\![T^c_{q_2}]\!]$.

PROOF: By induction on $|s|$. See Appendix B.1.2 for details.

Lemma 61 (Completeness of $\sim$). *If* $q_1 \not\sim q_2$ *then* $\mathcal{L}_{oi}[\![T^c_{q_1}]\!] \neq \mathcal{L}_{oi}[\![T^c_{q2}]\!]$.

PROOF: In duction on the iteration when q_1 and q_2 are separated. See Appendix B.1.3 for detail.

Now we are ready to prove that the algorithm produces a minimanl transducer.

Theorem 62 (Minimality of result). *If* $T' = (Q', \Sigma_i', \Sigma_o', q_i', Q_f', t')$ *is the result of applying the algorithm to an output-deterministic transducer* $T = (Q, \Sigma_i, \Sigma_o, q_i, Q_f, t)$, *then* T' *has exactly* $|\mathcal{S}[\![T]\!]|$ *nodes.*

PROOF: By combining the previous results. See Appendix B.1.4 for details.

We have now proved that the result of the algorithm without step 3 is a transducer that is equivalent under $=_{oi}$, and that the result is minimal under $=_{oi}$. This means that only reductions that changes the $\mathcal{L}_{oi}[\![T]\!]$ are missed when step 3 is skipped, and this suggests the reduced transducers to be close to minimal under $=_t$. In the following section we test the effect on the transducers and the subsequently generated enhanced DFAs to observe the effect on space and time consumption.

3.7 EMPIRICAL EVALUATION OF REDUCTION

We have added an implementation of the minimization algorithms – with and without step 3 – presented in Section 3.6 to the C++ library with the parser implementations [92], and used the original test machine to evaluate the effects of the reduction. The first application is to reduce the memory footprint of all the automata based implementations, namely DFA and DFASIM, due to the reduction of the represented automata. In the case of the DFA algorithm, this may render the algorithm feasible in the practical examples where it failed previously. We have measured the number of nodes in the generated transducers before and after the reductions, and the number of states used in the enhanced DFAs generated from the original transducer and the minimized transducers, for each of the practical examples [118]. The results are in the Figure 36. As the results show, there is a drastic reduction in the used number of transducer nodes as well as enhanced DFA states, and this means that the enhanced DFA construction is now feasible for all the examples. It is also worth noting that the difference in size between the reduction with and without step 3 is very small, and in most of the examples the algorithm without step 3 produces a transducer with one node less than when step 3 is included. This is not a bug, but a result of the assumption of no input-prefix for the initial node by step 3, that we solve by injecting a new initial node to the found prefix form of the transducer. This means that the result may have one node

Figure 36 The effects of the minimizations

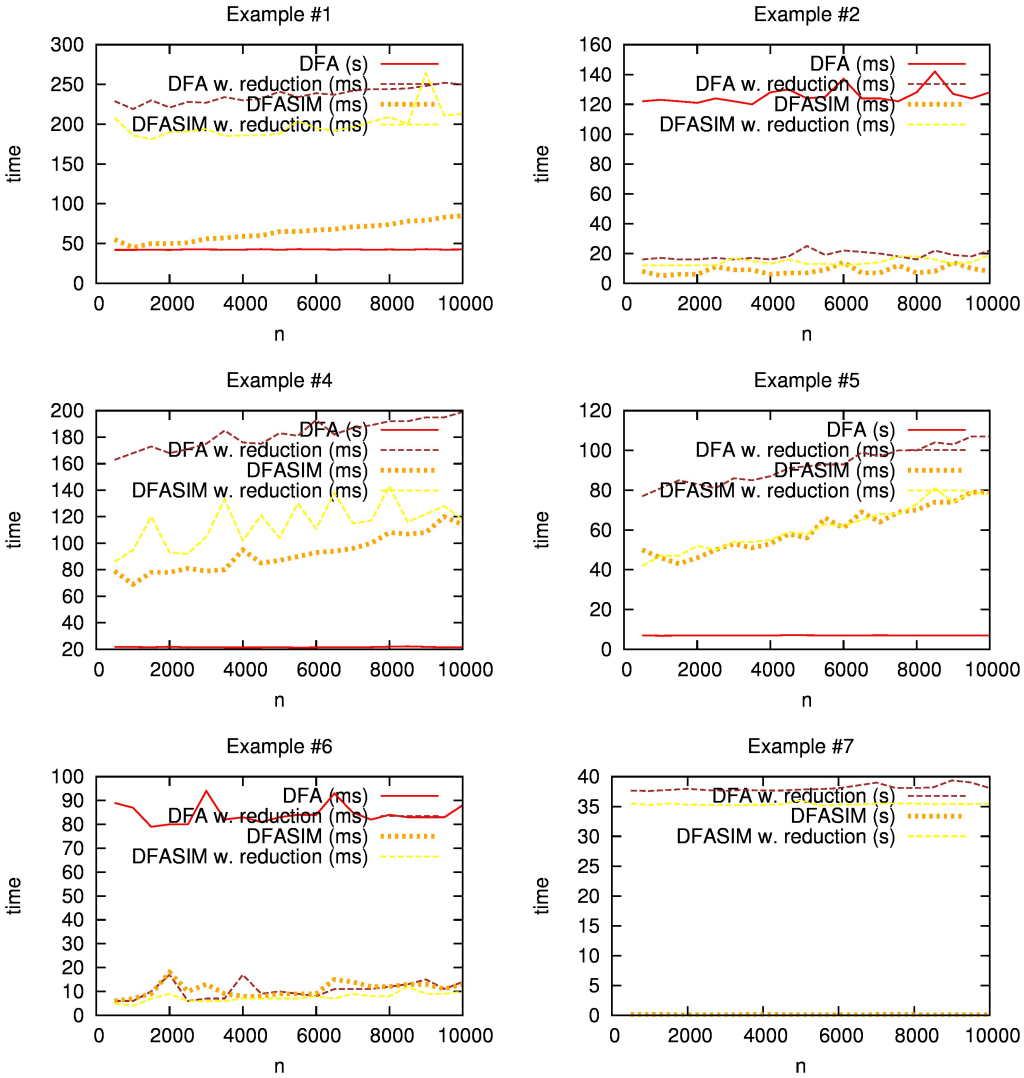
Test:	**NFA** **DFA**	**NFA-$=_{oi}$** **DFA-$=_{oi}$**	**DFA-$=_{t}$** **DFA-$=_{t}$**
Example #1	6077	203	204
	1847	8	9
Example #2	749	147	148
	133	13	13
Example #4	2365	358	359
	659	12	12
Example #5	2038	194	195
	388	6	6
Example #6	342	50	51
	89	8	8
Example #7	42185	5373	5304
	unknown	280	280
Example #8	20615	4789	4727
	unknown	141	141
Example #9	107836	6608	6609
	unknown	134	134
Example #10	6125	207	208
	unknown	12	13

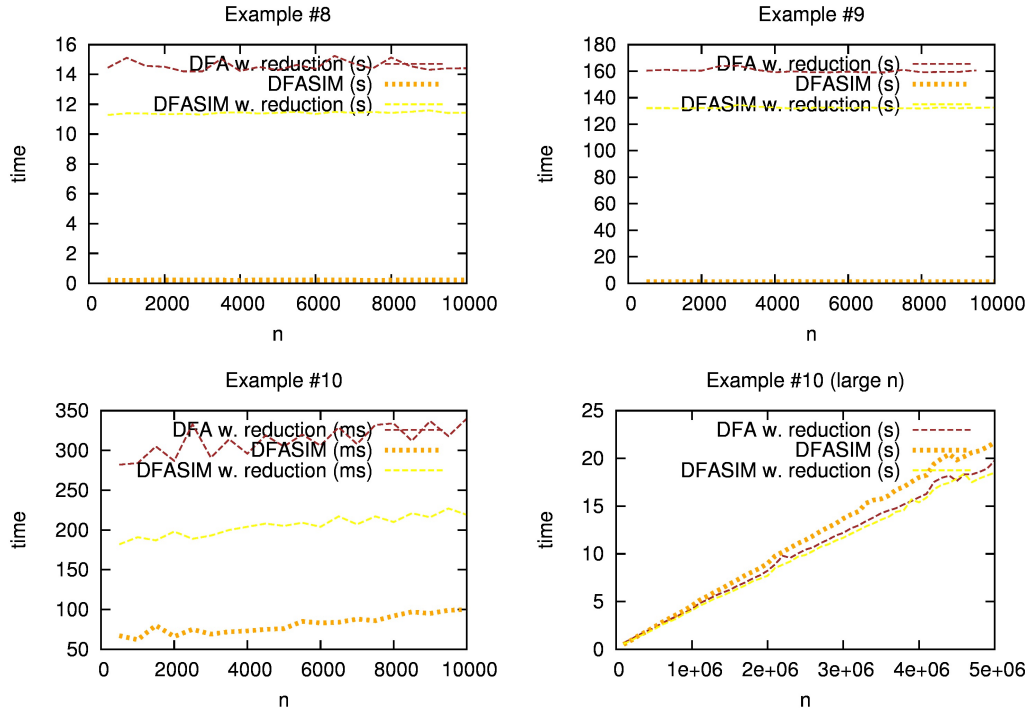
NFA and DFA holds the size of the unreduced transducer and the enhanced DFA it generates.
NFA-$=_{oi}$ and DFA-$=_{oi}$ holds the size of the $=_{oi}$-minimized transducer and the enhanced DFA it generates.
NFA-$=_{t}$ and DFA-$=_{t}$ holds the size of the $=_{t}$-minimized transducer and the enhanced DFA it generates.

more than the minimum, and in these cases the transducer found when skipping step 3 indeed has one node less and must thus be a minimal transducer (with respect to $=_{t}$). We can conclude that both minimizations significantly reduce the number of nodes in the transducer, and the number of nodes in the subsequently generated enhanced DFA. The effect on the size of the enhanced DFAs is twofold, as not only the number of nodes in the enhanced DFA is reduced, but each edge in the enhanced DFA contains a mapping for each transducer node in the source state, and as there are fewer transducer nodes, the size of each edge in the enhanced DFA is also reduced.

Another application is to improve the parsing time of the DFA and DFASIM

algorithms. The DFA algorithm can save time because it may take more time to generate the DFA from the unreduced transducer, than it does to perform the reduction and construct a DFA from the result. This is because the DFA generated from the original transducer may be significantly larger than the DFA generated from the reduced transducer. The DFA generated from the reduced transducer may parse more efficiently as well, because it is smaller and thus enjoys better memory locality. The DFASIM algorithm may become more efficient as well, because even though it does not generate the full DFA initially, it may have to generate less states during parsing by using the reduced transducer. The DFA and DFASIM algorithms have been executed on the practical examples using the transducer reduction, and the graphs below compare the used time to the execution without using transducer reduction.





As the results show, the DFA algorithm becomes a lot more efficient by using the transducer reduction, which means that the time it takes to generate the full DFA dominates the time it takes to reduce the transducer and generate the DFA for the reduced transducer. In most examples the DFASIM algorithm is more efficient without using the reduction. This is probably because only a few of the DFA states are generated when parsing the example strings, and therefore the gain of the reduction is dominated by the reduction time. The graphs do show that the DFASIM algorithm has a slower increase when the string length is increased when using the reduction, and therefore the DFASIM algorithm using the transducer reduction may be more efficient for very large strings, as demonstrated by the last graph where the input size is increased.

3.8 CONCLUSION

We have designed and implemented a number of regular expression parsing algorithms, which produce bit coded representations of parse trees without ever materializing the parse trees during parsing. Producing bit codings is advantageous since it carries the dual advantage of yielding a compressed parse tree representation and of speeding its construction. Our DFA simulation algorithm DFASIM, in the style of Dubé and Feeley [36], and FRCA, a modified version of the greedy algorithm of Frisch and Cardelli [46], have shown the best asymptotic

performance, with DFA simulation beating FRCA on a suite of real world examples. We have adapted Mohri's minimization algorithm for sequential transducers to the output-deterministic transducers generated. We have shown that the minimal transducer with respect to two differen equivalence relations are found depending on if step 3 is included in the algorithm. We have also studied the effects of the minimization algorithms on the sizes of the produced transducers and enhanced DFAs and the effect on the parsing efficiency. As for the potential for further improvements, efficient computation of the sub-transducers induced by an input string (left-to-right or right-to-left or, preferably, something better), and memoized DFA-state construction appear to be key to obtaining practically improved regular expression parsing without sacrificing asymptotic scalability.

Part III

SESSION TYPES

4

MULTIPARTY SYMMETRIC SUM TYPES

Authors:

Lasse Nielsen - University of Copenhagen,
Nobuko Yoshida - Imperial College London and
Kohei Honda - Queen Mary University of London

Presented at:

EXPRESS 2010 – 17th International Workshop on Expressiveness in Concurrency

This paper introduces a new theory of multiparty session types based on symmetric sum types, by which we can type non-deterministic orchestration choice behaviours. While the original branching type in session types can represent a choice made by a single participant and accepted by others determining how the session proceeds, the symmetric sum type represents a choice made by agreement among all the participants of a session. Such behaviour can be found in many practical systems, including collaborative workflow in healthcare systems for clinical practice guidelines (CPGs). Processes with the symmetric sums can be embedded into the original branching types using conductor processes. We show that this type-driven embedding preserves typability, satisfies semantic soundness and completeness, and meets the encodability criteria [51, 99] adapted to the typed setting. The theory leads to an efficient implementation of a prototypical tool for CPGs which automatically translates the original CPG specifications from a representation called the Process Matrix to symmetric sum types, type checks programs and executes them.

4.1 INTRODUCTION

Clinical Practice Guidelines (CPGs) [115] are detailed descriptions of medical treatment procedures, practised globally with local variations, in order to treat specific medical disorders. CPGs are an example of social interactions, which include workflow models and various cooperation models: its richness stems from the diverse collaborative patterns human organisations can exhibit. One such pattern, which plays a prominent role in CPGs, is *symmetric synchronisation* where all the participants are equal in the decision-making, i.e. the participants collectively decide on one of the possible choices.

Motivated from practice, this paper aims to distill the essence of this symmetric synchronisation as an interaction primitive, position it as part of the type theory for the asynchronous π-calculus with multiparty sessions, and explore its properties to model workflow frameworks, enjoying the richness of multiparty session types to express how data is exchanged. Our starting point is a widely known semi-formal modelling framework for CPGs and other workflows called Process Matrix [84], which provides a concise and general description of symmetric synchronisation patterns as found in CPGs.

The new synchronisation primitive is generally useful, also for other calculi and applications. We add the symmetric synchronisation primitive to the asynchronous π-calculus and study it in a typed setting because it allows us to model CPGs as types, and enables correctness and erasure properties.

We explain the key ideas of Process Matrix and CPGs using an example from a CPG with three participants: A doctor (D), a nurse (N) and a patient (P). The doctor and the nurse need to register and inspect the patient, thus they must obtain the patient data (Data), schedule an appointment (Schedule) and inspect the patient

Figure 37 Cases in the healthcare cooperation example

	Data	Schedule	Inspect
Case DD	D	D	D
Case ND	N	D	D
Case DN	D	N	D
Case NN	N	N	D

D: Doctor — Data: Obtain patient data
N: Nurse — Schedule: Schedule inspection
P: Patient — Inspect: Perform inspection

(Inspect). The actions can be divided between the doctor and the nurse in four different ways, since they both can collect the data and schedule the appointment but only the doctor may inspect the patient. The four cases are illustrated in the table in Fig. 37. For example in Case ND, the nurse obtains the patient data and the doctor schedules and performs the inspection. In this way, the doctor and the nurse need to perform a different combination of actions depending on which case is chosen, thus they need to commit to the same choice, in order for the cooperation to work. This cannot be ensured using the asymmetric choice (as found in branching/selection primitives in the foregoing session types [60, 114]), since the decision is done by a single participant and not by common agreement.

Our aim is to obtain a general modelling framework which can uniformly capture both symmetric synchronisations and existing session-based communication patterns. Such a framework will give a basis for the implementation of a tool for CPGs where one can describe, validate and execute specifications backed up by static validation coming from the theory. For this purpose we incorporate the synchronisation primitive in the type theory for multiparty sessions from [21, 61], so different groups of principals freely can mix standard asymmetric communications and symmetric synchronisations. The resulting sessions are abstracted as types, enabling type-based validation which ensures type and communication safety.

We offer the first prototype implementation of the π-calculus with multiparty sessions, with a typechecker using multiparty session types with full projections. Our implementation includes the symmetric synchronisation primitive and verification using symmetric sum types. This allows us to implement, verify and execute the examples used to explain and motivate the extension.

The use of types is not only essential for modelling CPGs and validating processes, but also enables an organised analysis of the synchronisation primitive. Using a type-directed translation, we show that the primitive can be embedded into the asymmetric branching in the original multiparty sessions [21, 61]. The translation generates auxiliary processes from the types, and combines them with an encoding of the sum into asymmetric branch types, respecting global interaction patterns and preserving semantics, by exploiting the type structure. The auxiliary process generated from a type conducts the synchronisations of a session by receiving accepted cases from participants and sending the chosen case back. To prove its correctness, we use a new technique based on derivations of the multiparty session typing. The resulting translation introduces exponentially more branching cases (e.g. 64 for the running example), demonstrating the practical usefulness of the symmetric sum for compact description as well as offering a formally founded distributed implementation strategy of the primitive.

Next we present the calculus for multiparty symmetric synchronisation (Section 4.2) and study its type theory (Section 4.3). We then define a type-directed encoding (Section 4.4) of the symmetric sum into the asynchronous multiparty

session; and investigate its encodability criteria by adapting the framework from [51, 99] to the typed setting. Finally we present an application of the theory to the formal CPG verification (Section 4.5), with a prototype implementation available from [1]. The technical contributions include *subject reduction* (Theorem 4.3.2) and *type/semantic correctness* of the encoding (Theorems 4.4.1, 4.4.2 and 4.4.4). The implementation demonstrates the correctness, feasible implementability and significance of the new primitive. In particular, an automatic mapping from Process Matrix to global types (Section 4.5) shows the expressiveness of multiparty session types. The omitted definitions, examples and proofs are included in Appendix refch:appendix:session:sumtypes.

Figure 38 The process language

$P ::= \mathtt{sync}_{\tilde{s},n}\{l : P_l\}_{l\in L}$	synchronisation	$\mid s \lhd l; P$	label selection
$\mid \mathtt{rand}\{P_i\}_{i\in I}$	random choice	$\mid s \rhd \{l : P_l\}_{l\in L}$	label branching
$\mid \overline{a}[2..n](\tilde{s}).P$	session request	$\mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	conditional
$\mid a[p](\tilde{s}).P$	session accept	$\mid P \mid Q$	parallel
$\mid s!\langle \tilde{e} \rangle; P$	value sending	$\mid 0$	inaction
$\mid s?(\tilde{x}); P$	value reception	$\mid (\nu n)P$	restriction
$\mid s!\langle\langle \tilde{s} \rangle\rangle; P$	delegation	$\mid \mathtt{def}\ D\ \mathtt{in}\ P$	recursion
$\mid s?((\tilde{s})); P$	reception	$\mid X\langle \tilde{e}\tilde{s} \rangle$	process call
		$\mid s : \tilde{h}$	message queue
$D ::= \{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I}$	declarations	$v ::= a \mid \mathtt{true} \mid \mathtt{false}$	values
$e ::= v \mid x \mid e\ \mathtt{and}\ e'$	expressions	$h ::= l \mid \tilde{v} \mid \tilde{s}$	messages
$\mid \mathtt{not}\ e \mid \mathtt{rand}\{v_i\}_{i\in I} \mid \ldots$			

4.2 PROCESSES WITH SYNCHRONISATION

This section introduces the syntax (Fig. 38) of the asynchronous multiparty session π-calculus [61] with the new sync primitive, and the judgement $P \to P'$ (Fig. 39, where $e \downarrow v$ denotes the evaluation of the expression e to the value v) describing the small-step semantics for processes. The syntax defines the values: $\{v, w, \ldots\}$, expressions: $\{e, e', \ldots\}$ and processes: $\{P, Q, \ldots\}$ from the sets of channel names: $\{a, b, \ldots\}$, value variables: $\{x, y, \ldots\}$, session channels: $\{s, t, \ldots\}$, labels: $\{l, m, \ldots\}$ and process variables: $\{X, Y, \ldots\}$.

Session request, $\overline{a}[2..n](\tilde{s}).P$ initiates a session with channels $\tilde{s}$ (where $\tilde{s}$ denotes a vector $s_1 \ldots s_n$) over the public channel a with the other $n-1$ participants of shape $a[p](\tilde{s}).Q_p$ for p from 2 to n ([Link] in Fig. 39). Asynchronous communication in an established session is performed by sending and receiving values ([Send,Recv]), transferring a session using session delegation and reception ([Deleg,SRec]), and label selection and branching ([Label,Branch]), where the branching process offers a number of labels and the selecting process chooses one of them.

The new $\mathtt{sync}_{\tilde{s},n}\{l : P_l\}_{l\in L}$ constructor is interpreted as the process participating in a plenum decision between all the n processes in the session $\tilde{s}$ reaching a common decision h from L. Afterwards the process proceeds as described in P_h. In [Sync] in Fig. 39, h in the premise denotes the common label. We also add the $\mathtt{rand}\{P_i\}_{i\in I}$ constructor which randomly selects one of its branches ([Rand]). This primitive can be expressed using if and a random expression (hence it does not add expressiveness from [61]), but simplifies the erasure mapping in Section 4.4.

In [Sync], the processes cannot perform the synchronisation if they do not share some common label, in which case the processes will be stuck. We also need to know how many participants are in the session in order to know when the synchronisation can step; otherwise the processes will be stuck. The typing system

introduced in the next section ensures that sync satisfies these two conditions.

HEALTHCARE COOPERATION (1): PROCESSES

We motivate the symmetric synchronisation using the example from the introduction. We first explain the problem when representing this interaction without sync. As explained in the introduction, there is no rigorous way to decide which of the four cases will occur, as well as who will be the principal decision maker: we could let the doctor non-deterministically decide between the cases, and then we obtain the processes in Fig. 40, if we are to use the processes from [61]: similarly we could let the nurse or even the patient decide. None of these representations captures the cooperation where the doctor, the nurse and the patient should reach a common decision, because *it is impossible to know who takes the initiative*. Another problem is that we need to specify the choices in P_D, which is best captured by non-deterministic expressions like rand.

Figure 39 The reduction rules

$$\frac{}{\overline{a}[2..n](\tilde{s}).P_1|a[2](\tilde{s}).P_2|\ldots|a[n](\tilde{s}).P_n \to (\nu\tilde{s})(P_1|P_2|\ldots|P_n|s_1:\varnothing|\ldots|s_m:\varnothing)}\ [\textsc{Link}]$$

$$\frac{\tilde{e}\downarrow\tilde{v}}{s!\langle\tilde{e}\rangle;P|s:\tilde{h}\to P|s:\tilde{h}\cdot\tilde{v}}\ [\textsc{Send}]$$

$$\frac{}{s?(\tilde{x});P|s:\tilde{v}\cdot\tilde{h}\to P[\tilde{v}/\tilde{x}]|s:\tilde{h}}\ [\textsc{Recv}] \qquad \frac{}{s\lhd l;P|s:\tilde{h}\to P|s:\tilde{h}\cdot l}\ [\textsc{Label}]$$

$$\frac{j\in I}{s\rhd\{l_i:P_i\}_{i\in I}|s:l_j\cdot\tilde{h}\to P_j|s:\tilde{h}}\ [\textsc{Branch}]$$

$$\frac{}{s!\langle\!\langle\tilde{t}\rangle\!\rangle;P|s:\tilde{h}\to P|s:\tilde{h}\cdot\tilde{t}}\ [\textsc{Deleg}] \qquad \frac{}{s?((\tilde{t}));P|s:\tilde{t}\cdot\tilde{h}\to P|s:\tilde{h}}\ [\textsc{SRec}]$$

$$\frac{e\downarrow \mathsf{true}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q\to P}\ [\textsc{IfT}] \qquad \frac{e\downarrow \mathsf{false}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q\to Q}\ [\textsc{IfF}]$$

$$\frac{\tilde{e}\downarrow\tilde{v}\quad X\langle\tilde{x}\tilde{s}\rangle=P\in D}{\mathsf{def}\ D\ \mathsf{in}\ X\langle\tilde{e}\tilde{s}\rangle|Q\to \mathsf{def}\ D\ \mathsf{in}\ P[\tilde{v}/\tilde{x}]|Q}\ [\textsc{Def}] \qquad \frac{P\to P'}{(\nu n)P\to(\nu n)P'}\ [\textsc{Scop}]$$

$$\frac{P\to P'}{P|Q\to P'|Q}\ [\textsc{Par}] \qquad \frac{P\to P'}{\mathsf{def}\ D\ \mathsf{in}\ P\to \mathsf{def}\ D\ \mathsf{in}\ P'}\ [\textsc{Defin}]$$

$$\frac{P\equiv P'\quad P'\to Q'\quad Q'\equiv Q}{P\to Q}\ [\textsc{Str}] \qquad \frac{j\in I}{\mathsf{rand}\{P_i\}_{i\in I}\to P_j}\ [\textsc{Rand}]$$

$$\frac{h\in\bigcap_{i=1}^{n}L_i}{\mathsf{sync}_{\tilde{s},n}\{l:P_{1l}\}_{l\in L_1}\mid\ldots\mid\mathsf{sync}_{\tilde{s},n}\{l:P_{nl}\}_{l\in L_n}\to P_{1h}\mid\ldots\mid P_{nh}}\ [\textsc{Sync}]$$

Figure 40 Healthcare example without `sync`

```
P_D = // Doctor
a[2](d,s,r,cp,cn).
if rand{true, false}
then cp◁CaseD;cn◁CaseD; d?(data); if rand{true, false}
      then cp◁CaseDD;cn◁CaseDD;s!⟨eSchedule⟩;r!⟨eResult⟩;end
      else cp◁CaseDN;cn◁CaseDN;r!⟨eResult⟩;end
else cp◁CaseN;cn◁CaseN; if rand{true, false}
      then cp◁CaseND;cn◁CaseND;s!⟨eSchedule⟩;r!⟨eResult⟩;end
      else cp◁CaseNN;cn◁CaseNN;r!⟨eResult⟩;end

P_P = // Patient
ā[2..3](d,s,r,cp,cn). pd▷
{CaseD: d!⟨eData⟩; cp▷
   {CaseDD: s?(schedule);r?(result);end,
    CaseDN: s?(schedule);r?(result);end },
 CaseN: d!⟨eData⟩; pd▷
   {CaseND: s?(schedule);r?(result);end,
    CaseNN: s?(schedule);r?(result);end }
}

P_N = // Nurse
a[3](d,s,r,cp,cn). cn▷
{CaseD: cn▷ {CaseDD: end, CaseDN: s!⟨eSchedule⟩;end },
 CaseN: d?(data); cn▷ {CaseND: end,
                       CaseNN: s!⟨eSchedule⟩;end }
}
```

Fig. 41 describes the same example using `sync` where the intended cooperation is directly modelled. The case is logically decided by two choices: first it is decided who receives the patient data, and then it is decided who schedules the inspection. Since these decisions are not necessarily made at the same time, the processes select the case using two sequential synchronisations.

4.3 SYMMETRIC SUM TYPES

We start by defining the global types G in Fig. 42, which specifies global session protocols between the participants. Except for the symmetric sum type, the syntax is from [61]. The type $\mathtt{p} \to \mathtt{p}' : \mathtt{k}\langle \mathtt{U} \rangle.\mathtt{G}'$ expresses that participant p sends a message

of type U along channel k to p' and then interactions described in G' take place. The type $p \to p' : k\{l_i : G_i\}_{i \in I}$ expresses that p sends one of the labels l_i to p'. If l_j is sent, interactions described in G_j take place. Type $\mu t.G$ is a recursive type, assuming type variables $(t, t', \ldots)$ are guarded in the standard way. We assume that G in the grammar of sorts is closed, i.e., without free type variables. Type end represents the session termination.

The sum type $\{l : G_l\}_{l \in L;M}$ represents a synchronisation where the labels are taken from the set L and the non-empty set M. The labels in L are optional, but the labels in M are mandatory and must be accepted by all the participants. The mandatory labels will be underlined to distinguish them from the optional labels (e.g. $\{l : G_l\}_{l \in \{l1\};\{l2\}} = \{l1 : G_{l1}, \underline{l2} : G_{l2}\}$).

The local types T are defined in Fig. 42. They describe the communication performed by a single process. Therefore the "from process to process on channel"

Figure 41 Healthcare example using sync

```
P_D = // Doctor
a[2](d,s,r). sync((d,s,r),3)
{CaseD: d?(data); sync((d,s,r),3)
   {CaseDD: s!⟨eSchedule⟩;r!⟨eResult⟩;end,
    CaseDN: r!⟨eResult⟩;end },
 CaseN: sync((d,s,r),3)
   {CaseND: s!⟨eSchedule⟩;r!⟨eResult⟩;end,
    CaseNN: r!⟨eResult⟩;end } }

P_P = // Patient
ā[2..3](d,s,r). sync((d,s,r),3)
{CaseD: d!⟨eData⟩; sync((d,s,r),3)
   {CaseDD: s?(schedule);r?(result);end,
    CaseDN: s?(schedule);r?(result);end },
 CaseN: d!⟨eData⟩; sync((d,s,r),3)
   {CaseND: s?(schedule);r?(result);end,
    CaseNN: s?(schedule);r?(result);end } }

P_N = // Nurse
a[3](d,s,r). sync((d,s,r),3)
{CaseD: sync((d,s,r),3)
   {CaseDD: end, CaseDN: s!⟨eSchedule⟩;end },
 CaseN: d?(data); sync((d,s,r),3)
   {CaseND: end, CaseNN: s!⟨eSchedule⟩;end } }
```

Figure 42 The Domains used for Global and Local types

GLOBAL TYPES:

$$G ::= p \to p' : k\langle U\rangle.G' \mid p \to p' : k\{l_i : G_i\}_{i\in I} \mid \mu t.G \mid t \mid \mathsf{end}$$
$$\mid \{l : G_l\}_{l\in L;M}\ (M \neq \varnothing)$$

LOCAL TYPES:

$$T ::= k!\langle U\rangle;T \mid k?\langle U\rangle;T \mid k \oplus \{l : T_l\}_{l\in L} \mid k \,\&\, \{l : T_l\}_{l\in L} \mid \mu t.T \mid t \mid \mathsf{end}$$
$$\mid \{l : T_l\}_{l\in L;M}\ (M \neq \varnothing)$$

MESSAGE TYPES: $U ::= \tilde{S} \mid T@(p, m, n)$

SIMPLE TYPES: $S ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots \mid \langle G\rangle$

ENVIRONMENTS:

$$\Gamma ::= \varnothing \mid \Gamma, u : \langle G\rangle \mid \Gamma, X : \tilde{S}\tilde{T} \qquad \Delta ::= \varnothing \mid \Delta, \tilde{s} : T@(p, n)$$

syntax is simply changed to sending or receiving on a channel. Thus the sending type is $k!\langle U\rangle;T$ and represents sending a message of type U on channel k, followed by the communication described by T. The type of receiving is $k?\langle U\rangle;T$, the type of selecting is $k \oplus \{l : T_l\}_{l\in L}$ and the type of branching is $k \,\&\, \{l : T_l\}_{l\in L}$. The difference from [61] is that the symmetric sum type constructor $\{l : T_l\}_{l\in L;M}$ is added where L, M satisfies the conditions similar to those of global type.

The message type $T@(p, m, n)$ is used for delegation. It describes an open session, and includes information about the participant number p, the number of session channels m, and the number of participants n in the session together with a local type T describing the remaining communication.

Finally we define the global environment Γ containing the global types for shared channels u, and process variables X, and the local type environment Δ containing the remaining session communication in Fig. 42, where $\tilde{s} : T@(p, n)$ means $\tilde{s}$ is an open session with n participants, where T describes the remaining communication for participant p.

The *projection* $G \upharpoonright p$ of a global type G for a participant p generates the local type for the participant in an intuitive way, for example $(p_0 \to p_1 : k\langle U\rangle.G') \upharpoonright p$ becomes $k!\langle U\rangle;(G' \upharpoonright p)$ if $p = p_0$ and $p \neq p_1$. The differences from the definition in [61] is that we have added a case for the symmetric sum type, $(\{l : G_l\}_{l\in L;M}) \upharpoonright p = \{l : (G_l \upharpoonright p)\}_{l\in L;M}$.

A global type G is coherent [61] if and only if the projection $G \upharpoonright p$ is defined for all participants, and G does not allow racing conditions (linearity). We only consider coherent global types.

Figure 43 Selected typing rules

$$\frac{[\textsc{Rand}] \quad \forall i \in I. \Gamma \vdash P_i \rhd \Delta \quad I \neq \varnothing}{\Gamma \vdash \mathsf{rand}\{P_i\}_{i\in I} \rhd \Delta}$$

$$\frac{[\textsc{Sync}] \quad \forall l \in L'' : \Gamma \vdash P_l \rhd \Delta, \tilde{s} : T_l@(p,n) \quad L'' \subseteq L \cup L' \quad L' \subseteq L''}{\Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{l : P_l\}_{l\in L''} \rhd \Delta, \tilde{s} : \{l : T_l\}_{l\in L;L'}@(p,n)}$$

$$\frac{\begin{array}{c}|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \quad n = \mathsf{max}(\mathsf{pid}(G)) \\ [\textsc{Mcast}] \quad \Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \rhd \Delta, \tilde{s} : (G\upharpoonright 1)@(1,n)\end{array}}{\Gamma \vdash \overline{a}[2\,..\,n](\tilde{s}).P \rhd \Delta}$$

$$\frac{\begin{array}{c}|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \quad n = \mathsf{max}(\mathsf{pid}(G)) \\ [\textsc{Macc}] \quad \Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \rhd \Delta, \tilde{s} : (G\upharpoonright p)@(p,n)\end{array}}{\Gamma \vdash a[p](\tilde{s}).P \rhd \Delta}$$

$$\frac{[\textsc{Send}] \quad \forall j. \Gamma \vdash e_j : S_j \quad \Gamma \vdash P \rhd \Delta, \tilde{s} : T@(p,n)}{\Gamma \vdash s_k!\langle \tilde{e} \rangle; P \rhd \Delta, \tilde{s} : k!\langle \tilde{S} \rangle; T@(p,n)}$$

$$\frac{[\textsc{Rcv}] \quad \Gamma, \tilde{x} : \tilde{S} \vdash P \rhd \Delta, \tilde{s} : T@(p,n)}{\Gamma \vdash s_k?(\tilde{x}); P \rhd \Delta, \tilde{s} : k?\langle \tilde{S} \rangle; T@(p,n)}$$

$$\frac{[\textsc{Sel}] \quad \Gamma \vdash P \rhd \Delta, \tilde{s} : T_h@(p,n) \quad h \in L}{\Gamma \vdash s_k \lhd h; P \rhd \Delta, \tilde{s} : k \oplus \{l : T_l\}_{l\in L}@(p,n)}$$

$$\frac{[\textsc{Branch}] \quad \forall l \in L : \quad \Gamma \vdash P_l \rhd \Delta, \tilde{s} : T_l@(p,n)}{\Gamma \vdash s_k \rhd \{l : P_l\}_{l\in L} \rhd \Delta, \tilde{s} : k?\{l : T_l\}_{l\in L}@(p,n)}$$

$$\frac{[\textsc{Conc}] \quad \Gamma \vdash P \rhd \Delta \quad \Gamma \vdash Q \rhd \Delta'}{\Gamma \vdash P|Q \rhd \Delta \circ \Delta'} \; (\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \varnothing)$$

JUDGEMENT The typing judgement extends the one from [61] with symmetric sum types. The judgement $\Gamma \vdash P \rhd \Delta$ states that the process P in the environment Γ performs exactly the session communication described in Δ.

The main rules are included in Fig. 43. The local types now carry information about the number of participants n and channels m. The number of participants and channels is determined at the session initialisation in the rules [MCAST] and [MACC], where $\mathsf{sid}(G)$ denotes channels that appear in G and $\mathsf{pid}(G)$ denotes the participants that appear in G. The rule [SYNC] checks that the accepted branches in a synchronisation includes the mandatory ones and does not exceed the optional ones, and checks that each accepted branch is typed with the correct communication. The typing rule [RAND] checks that each choice in a `rand` process has the same session environment.

Since the process is reduced by each rule-application, the typability question $\Gamma \vdash P \triangleright \Delta$ is decidable.

HEALTHCARE COOPERATION (2): TYPES

We explain how the types can describe and verify the healthcare scenario in the Introduction. Recall the processes from Fig. 41. To type $P_D \mid P_N \mid P_P$, we need a matching type-environment first. The processes use the public channel a to create a session, so the environment must be of the form $\Gamma = a : \langle G \rangle$ for some global type G.

We will start by finding the type describing the interactions in CaseND. First the participants select the choice CaseN and the patient sends the data to the nurse. Then the participants select the choice CaseND, the doctor sends the schedule to the patient, and finally the doctor sends the result to the patient.

When the patient has id 1, the doctor has id 2 and the nurse has id 3 the described communication for CaseND is described by the type

$$\{\underline{\mathsf{CaseN}}\colon\ 1{\to}3{:}1\langle \mathsf{Sdata}\rangle.\ \{\underline{\mathsf{CaseND}}\colon\ 2{\to}1\colon\ 2\langle \mathsf{Sschedule}\rangle.\ 2{\to}1\colon\ 3\langle \mathsf{Sresult}\rangle.\ \mathsf{end}\}\ \}$$

Performing the same reasoning for CaseDD, CaseDN and CaseNN and adding their branches to the symmetric sums results in the global type G in Fig. 44. We select CaseND, CaseDN and CaseN as the mandatory labels. Since all participants must accept the mandatory choices, this means that it is always possible for the participants to agree on a choice in each of the synchronisations. We can then find the local type for the patient process as the patient's projection of G, given in Fig. 44. Using this type and the projections we can now typecheck the processes (see Proof C.10.1).

Proposition 4.3.1. $a : \langle G \rangle \vdash P_D \mid P_N \mid P_P \triangleright \varnothing$.

We end this section by proving *subject reduction*, from which we can derive soundness, communication safety and progress [61, § 5] as corollaries. Below $\Delta \to^{0/1} \Delta'$ denotes zero or one step using the type reduction [61], which represents

the communication between dual local types. For instance, a reduction between input and output types is defined as:

$$k!\langle U\rangle;T_1@(p,n),k?\langle U\rangle;T_2@(q,n) \to T_1@(p,n),T_2@(q,n).$$

We extend it to the symmetric sum as:

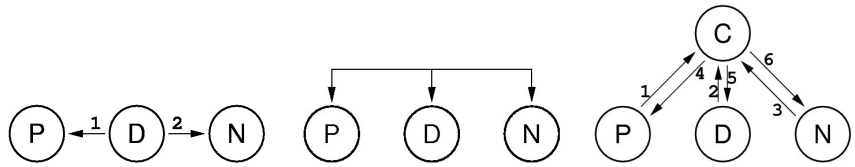
$$\{\{l:T_p,\ldots\}@(p,n)\}_{p\in\{1..n\}} \to \{T_p@(p,n)\}_{p\in\{1..n\}}.$$

The formulation uses the extension of the typing to runtime processes ($\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$), which corresponds to the presented typing on processes without open sessions, but also accept processes with open sessions. This is obtained by joining compatible session environments (Δ, Δ') using the $\Delta \circ \Delta'$ operation to a single environment expressing the communication in both Δ and Δ'. Then we have:

Figure 44 Global Type G and patient projection for healthcare example

```
G = // Global type
{CaseD:
  1→2:1⟨Sdata⟩;
  {CaseDD: 2→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end,
   CaseDN: 3→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end
  },
 CaseN:
  1→3:1⟨Sdata⟩;
  {CaseND: 2→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end,
   CaseNN: 3→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end
  }
}

G↑1 = // Local type for Patient
{CaseD:
  1!⟨Sdata⟩;
  {CaseDD: 2?⟨Sschedule⟩;3?⟨Sresult⟩;end,
   CaseDN: 2?⟨Sschedule⟩;3?⟨Sresult⟩;end
  },
 CaseN:
  1!⟨Sdata⟩;
  {CaseND: 2?⟨Sschedule⟩;3?⟨Sresult⟩;end,
   CaseNN: 2?⟨Sschedule⟩;3?⟨Sresult⟩;end
  }
}
```

Figure 45 Synchronisation message flows



Theorem 4.3.2 (Subject reduction)**.**
If $\Gamma \vdash P \rhd_{\tilde{s}} \Delta$, Δ *coherent and* $P \to P'$ *then* $\Gamma \vdash P' \rhd_{\tilde{s}} \Delta'$ *where* $\Delta \to^{0/1} \Delta'$.

PROOF: By induction on the derivation of $P \to P'$. See Proof C.3.2.

4.4 FROM SYMMETRIC SUM TO CONDUCTED BRANCHING

This section studies an erasure of symmetric synchronisation, which translates away symmetric sums using existing session primitives, which we hereafter simply call *the erasure*. The erasure removes all occurrences of the `sync` constructor while preserving static and dynamic semantics, i.e. typability and reduction. It uses a conductor process for each session. The messages and protocol used to implement the synchronisation are illustrated in Fig. 45 where the numbers indicate the sequence of the messages. Fig. 45(a) shows the communication between the processes without using `sync` in Fig. 40. Fig. 45(b) shows the communication between the processes using `sync` in Fig. 41, where no messages are sent, because the synchronisation ensures the same branch is chosen. Fig. 45(c) shows the conduction messages in the processes where the synchronisation has been erased in Fig. 49. First the patient, the doctor and the nurse send the cases they can accept to the conductor, who chooses a common case and sends the selected case to the patient, the doctor and the nurse.

4.4.1 *Erasure definitions*

Based on this idea, we translate the synchronisation and symmetric sum types into the original system [61], step by step as follows.

STEP 1: PROCESS ERASURE Only well-typed processes are eligible for erasure, because conductor processes are generated from the global types. Therefore the erasure $\mathcal{E}[\![\cdot]\!]$ is defined on the type derivation in Fig. 46 and the result is the erased process. We use the notation $\mathcal{D} :: \Gamma \vdash P \rhd \Delta$ to denote a derivation $\mathcal{D}$ with the conclusion $\Gamma \vdash P \rhd \Delta$.

Figure 46 Erasure of Synchronisation from Typing-Derivation

$$\mathcal{E}\left[\!\!\left[\frac{[\textsc{Mcast}] \quad \Gamma \vdash \overline{a}[2..n](\tilde{s}).P \triangleright \Delta}{\Gamma \vdash \overline{a}[2..n](\tilde{s}).P \triangleright \Delta} \right]\!\!\right] = \mathcal{C}\,[\![G]\!]_{\tilde{s},n,a} \mid \overline{a}[2..n,n{+}1](\tilde{s},\mathtt{in}_{\tilde{s}1},\mathtt{out}_{\tilde{s}1},\ldots,\mathtt{in}_{\tilde{s}n},\mathtt{out}_{\tilde{s}n}).\mathcal{E}\,[\![\mathcal{D}_1]\!]$$

$$\mathcal{E}\left[\!\!\left[\frac{[\textsc{Macc}] \quad \Gamma \vdash a[p](\tilde{s}).P \triangleright \Delta}{\Gamma \vdash a[p](\tilde{s}).P \triangleright \Delta} \right]\!\!\right] = a[p](\tilde{s},\mathtt{in}_{\tilde{s}1},\mathtt{out}_{\tilde{s}1},\ldots,\mathtt{in}_{\tilde{s}n},\mathtt{out}_{\tilde{s}n},).\mathcal{E}\,[\![\mathcal{D}_1]\!]$$

$$\mathcal{E}\left[\!\!\left[\frac{[\textsc{Sync}] \quad \Gamma \vdash \mathtt{sync}_{\tilde{s},n}\{l : P_l\}_{l\in L'} \triangleright \Delta, \tilde{s} : \{l : T_l\}_{l\in L;M,n}@(p,n)}{\Gamma \vdash \mathtt{sync}_{\tilde{s},n}\{l : P_l\}_{l\in L'} \triangleright \Delta, \tilde{s} : \{l : T_l\}_{l\in L;M,n}@(p,n)} \right]\!\!\right] = \mathtt{out}_{\tilde{s}p} \lhd \mathtt{cases}_{L'}; \mathtt{in}_{\tilde{s}p} \rhd \{l : \mathcal{E}\,[\![\mathcal{D}_l]\!]\}_{l\in L'}$$

$$\mathcal{E}\left[\!\!\left[\frac{[\textsc{Label}] \quad \Gamma \vdash s_k \lhd h; P \triangleright \Delta, \tilde{s} : k \oplus \{l : T_l\}_{l\in L}@(p,n)}{\Gamma \vdash s_k \lhd h; P \triangleright \Delta, \tilde{s} : k \oplus \{l : T_l\}_{l\in L}@(p,n)} \right]\!\!\right] = s_k \lhd h; \mathtt{out}_{\tilde{s}p} \lhd h; \mathcal{E}\,[\![\mathcal{D}_1]\!]$$

The other cases are monomorphically defined

The case for session request increments the number of participants by one, to make room for the conductor process, and adds two session channels per user ($\mathtt{in}_{\tilde{s},p}$ and $\mathtt{out}_{\tilde{s},p}$), for communicating with the conductor. The conductor process $\mathcal{C}\,[\![G]\!]_{\tilde{s},n,a}$ (defined in Step 2) is inserted in parallel with the resulting session requesting process to ensure it is available.

The case for synchronisation sends the accepted labels to the conductor, waits to receive one of the accepted labels and proceeds with the selected branch.

STEP 2: CONDUCTOR GENERATION The conductor process $\mathcal{C}\,[\![G]\!]_{\tilde{s},n,a}$ was inserted in parallel with the session requests by the process erasure in Step 1. The main cases of the conductor generation $\mathcal{C}\,[\![\cdot]\!]$ are in Fig. 47. Notice that $\mathcal{C}\,[\![G]\!]_{\tilde{s},n,a}$ is only a wrapper for $\mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n}$ which prefixes the session acceptance on channel a. In $\mathcal{C}\,[\![G]\!]_{\tilde{s},n,a}$, $\tilde{s}$ is the original session channels, n is the number of original participants, G is the original session type, and a is the channel the session is created over.

The conductor process generated from a synchronisation receives the accepted labels from each participant, selects a common label using `rand` and sends the

Figure 47 Conductor Process Generation from a Global Type

$$\begin{aligned}
\mathcal{C}\,[\![G]\!]_{\tilde{s},n,a} &= a[n{+}1](\tilde{s},\mathtt{in}_{\tilde{s},1},\mathtt{out}_{\tilde{s},1},\ldots,\mathtt{in}_{\tilde{s},n},\mathtt{out}_{\tilde{s},n}).\mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n} \\
\mathcal{C}\,[\![\{l : G_l\}_{l\in L;M}]\!]^{\star}_{\tilde{s},n} &= \mathtt{out}_{\tilde{s}1} \rhd \{\mathtt{cases}_{L_1\cup M} : \ldots : \mathtt{out}_{\tilde{s}n} \rhd \{\mathtt{cases}_{L_n\cup M} : \\
&\quad \mathtt{rand}\{\mathtt{in}_{\tilde{s}1} \lhd l; \ldots; \mathtt{in}_{\tilde{s}n} \lhd l; \mathcal{C}\,[\![G_l]\!]^{\star}_{n,\tilde{s}}\}_{l\in\bigcap_{i=1}^{n} L_i\cup M}\}_{L_n\subseteq L}\ldots\}_{L_1\subseteq L}
\end{aligned}$$

Figure 48 Erasure Mapping for Global Types

$$
\begin{array}{rcl}
[\![\mathsf{G}]\!] & = & [\![\mathsf{G}]\!]^{\star}_{\mathsf{max}(\mathsf{pid}(\mathsf{G})),\mathsf{max}(\mathsf{sid}(\mathsf{G}))} \\
[\![\{l : \mathsf{G}_l\}_{l\in L;M}]\!]^{\star}_{n,m} & = & 1 \to n{+}1 : (m+2)\{cases_{L_1\cup M} : \\
 & & 2 \to n{+}1 : (m+4)\{cases_{L_2\cup M} : \ldots \\
 & & n \to n{+}1 : (m+2\cdot n)\{cases_{L_n\cup M} : \\
 & & n{+}1 \to 1 : (m+1)\{l : n{+}1 \to 2 : (m+3)\{l : \ldots \\
 & & n{+}1 \to n : (m+2\cdot n-1)\{l : [\![\mathsf{G}_l]\!]^{\star}_{n,m}\}\ldots\} \\
 & & \}_{l\in\bigcap_{i=0}^{n} L_i\cup M}\}_{L_n\subseteq L}\cdots\}_{L_1\subseteq L}
\end{array}
$$

selected label back to each participant before conducting the chosen branch.

STEP 3: TYPE TRANSLATIONS To prove that typability is preserved by the erasure, we define translations of global types, local types, message types, global type environments and local type environments to find the types for the result of the erasure. The main cases for global types are defined in Fig. 48. The translation $[\![\mathsf{G}]\!]$ of global types is just a wrapper for $[\![\mathsf{G}]\!]^{\star}_{n,m}$ where n is the number of participants, and m is the number of session channels in the original type.

As previously suggested, the symmetric sum is translated to nested branching, where each participant sends the accepted labels to the conductor, receives the selected label and continues with the selected branch.

4.4.2 *Correctness*

We now prove the correctness of the erasure mapping. We start by proving that the typing is preserved, and the types of the result process is given by the defined type translations.

Theorem 4.4.1 (Type preservation)**.** *If* $\mathcal{D} :: \Gamma \vdash P \triangleright \Delta$ *then* $[\![\Gamma]\!] \vdash \mathcal{E}\,[\![\mathcal{D}]\!] \triangleright [\![\Delta]\!]$

PROOF: By induction on the type derivation $\mathcal{D}$. See Proof C.5.4. The proof uses a lemma stating that the generated conductor processes are well-typed.

Next we prove that process congruence ($P \equiv Q$) is preserved by the erasure.

Theorem 4.4.2 (Congruence preservation)**.**
If $\mathcal{D}_1 :: \Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ *then for all* Q *we have that* $P \equiv Q$ *if and only if there is a derivation* $\mathcal{D}_2 :: \Gamma \vdash Q \triangleright_{\tilde{t}} \Delta$ *such that* $\mathcal{E}\,[\![\mathcal{D}_1]\!] \equiv \mathcal{E}\,[\![\mathcal{D}_2]\!]$.

PROOF: See Proof C.6.1 and Proof C.6.5.

Congruence preservation suggests the erasure preserves semantic properties. We start by stating the soundness theorem. To do this we define conductors for partially completed sessions: $\mathtt{PC}(\Delta)$ as the set of possible partial conductor processes

generated from Δ. By using the partial conductors from the session environment it is now possible to state the soundness theorem.

Theorem 4.4.3 (Soundness). *If* $\mathcal{D} :: \Gamma \vdash P \triangleright_{\tilde{t}} \Delta$, $P \to P'$, Δ *coherent and* $P_C \in \mathsf{PC}(\Delta \circ \Delta'')$ *for some* Δ'' *then there is a derivation* $\mathcal{D}' :: \Gamma \vdash P' \triangleright_{\tilde{t}} \Delta'$ *and* $P'_C \in \mathsf{PC}(\Delta' \circ \Delta'')$ *such that* $\Delta \to^{0/1} \Delta'$ *and* $\mathcal{E}\,[\![\mathcal{D}]\!]\,|P_C \to^\star \mathcal{E}\,[\![\mathcal{D}']\!]\,|P'_C$.

Proof: By induction on the derivation of $P \to P'$.

We can extend the above theorem to multiple steps by induction on the number of steps (Corollary C.7.3). Also the found evaluation of $\mathcal{E}\,[\![\mathcal{D}]\!] \to^\star \mathcal{E}\,[\![\mathcal{D}']\!]$ performs exactly the same communication on all non-conductor channels as the original evaluation $P \to^\star P'$.

We will now define *conduction steps*, since they play an important role in formulating the completeness theorem. This is because all steps performed by the result of the erasure can be mimicked by the original process up to *conduction steps*. A step from P_1 to P_2 is a conduction step, written $P_1 \rightarrowtail P_2$ if the step performs *label selection* or *label branching* on a conductor channel or unfolding of a *conductor process*; otherwise we write $P_1 \twoheadrightarrow P_2$. We observe all the extra steps introduced by the erasure are of the form $\rightarrowtail$, while the other steps are of the form $\twoheadrightarrow$. Therefore there is a one-to-one correspondence between the $\twoheadrightarrow$ steps of the erased process, and the steps in the original process.

Theorem 4.4.4 (Semantic completeness). *If* $\mathcal{E}\,[\![\mathcal{D}_1 :: \Gamma \vdash P_1 \triangleright \varnothing]\!] \to^\star Q'$ *then there exists a derivation* $\mathcal{D}_2 :: \Gamma \vdash P_2 \triangleright \varnothing$ *and* Q *such that* $P_1 \to^\star P_2$ *and* $\mathcal{E}\,[\![\mathcal{D}_2]\!] \rightarrowtail^\star Q$ *and* $Q' \rightarrowtail^\star Q$.

Proof: By induction on the number of non-conduction steps in $\mathcal{E}\,[\![\mathcal{D}_1]\!] \to^\star Q'$, using confluence and single-step completeness results. See Proof C.8.12.

HEALTHCARE COOPERATION (3): SYNCHRONISATION ERASURE

The result of the erasure on the healthcare example from Section 4.3 is shown in Fig. 49. Since we have showed that the processes from the synchronisation example in Fig. 41 are well-typed in Proposition 4.3.1, we can apply Theorem 4.4.1 to provide $a : \langle [\![G]\!] \rangle \vdash P'_C \mid P'_P \mid P'_D \mid P'_N \triangleright \varnothing$.

As this example illustrates, the result of the erasure does not capture the nature of the situation in the same way, because it introduces a conductor process, which is not a natural part of the situation. It is not compact either, as the conductor process has 64 cases. Further we lose an accurate type abstraction of the dynamics of symmetric synchronisation, because it is not clear from the encoded type structure whether it is just a sequence of asymmetric branching actions or the (intended) atomic multiparty synchronisation, since some of the key operational structures of the encoding (e.g. random selection) is lost in the encoded type.

Figure 49 Example Processes after Erasure

```
P'_C = // Conductor
a[4](d,s,r,in_p,out_p,
     in_d,out_d,in_n, out_n).
out_p ▷
{cases_DN: out_d ▷
 {cases_DN: out_n ▷
  {cases_DN:
   rand {
    in_p ◁ CaseD;
    in_d ◁ CaseD;
    in_n ◁ CaseD;
    out_p ▷
    {cases_DN: out_d ▷
     {cases_DN: out_n ▷
      {cases_DN:
       rand
       {in_p ◁ CaseDD;
        in_d ◁ CaseDD;
        in_n ◁ CaseDD;
       end,
        in_p ◁ CaseDN;
        in_d ◁ CaseDN;
        in_n ◁ CaseDN;
       end },
      cases_D: ... },
     cases_D: ... },
    cases_D: ... } },
  cases_N: ... },
 cases_N: ... },
cases_N: ... }
```

```
P'_P = // Patient
ā[2..4](d,s,r, in_p, out_p,
   in_d, out_d, in_n, out_n).
out_p◁cases_DN;in_p▷
{CaseD: d!⟨Data⟩;out_p◁cases_DN;in_p▷
 {CaseDD: s?(schedule);r?(result);o,
  CaseDN: s?(schedule);r?(result);o},
 CaseN: d!⟨Data⟩;out_p◁cases_DN;in_p▷
 {CaseND: s?(schedule);r?(result);o,
  CaseNN: s?(schedule);r?(result);o} }
P'_D = // Doctor
a[2](d,s,r, in_p, out_p,
   in_d, out_d, in_n, out_n).
out_d◁cases_DN;in_d▷
{CaseD: d?⟨data⟩;out_d◁cases_DN;in_d▷
 {CaseDD: s!⟨Schedule⟩;r!⟨Result⟩;o,
  CaseDN: r!⟨Result⟩;o},
 CaseN: out_d◁cases_DN;in_d▷
 {CaseND: s!⟨Schedule⟩;r?⟨Result⟩;o,
  CaseNN: r?⟨Result⟩;o} }
P'_N = // Nurse
a[3](d,s,r, in_p, out_p,
   in_d, out_d, in_n, out_n).
out_n◁cases_DN;in_n▷
{CaseD: out_n◁cases_DN;in_n▷
 {CaseDD: o,
  CaseDN: s!⟨Schedule⟩;o},
 CaseN: d?⟨data⟩;out_n◁cases_DN;in_n▷
 {CaseND: o,
  CaseNN: s!⟨Schedule⟩;o} }
```

4.4.3 *Encodability criteria*

The common properties of encodability from the known separation theorems (e.g. [99]) has been studied [51], revealing a number of desirable criteria. Our encoding is *type-based*, so we cannot apply this untyped framework directly. However if we simply change the formulation to use the *type-derivation* instead of the process syntax, our encoding *does* fulfil the criteria.

Before we can define and prove the criteria, we need to define the relations ($\asymp_1$ and $\asymp_2$) and properties (successful state) used to define the criteria. We select $\asymp_1$ as the process equivalence ($\equiv$), and define $Q_1 \asymp_2 Q_2$ if and only if $\exists Q.Q_1 \rightarrow^\star Q \ \wedge \ Q_2 \rightarrow^\star Q$.

Lemma 4.4.5. *$\asymp_2$ is a weak barbed reduction congruence.*

Proof: Immediately $\asymp_2$ is symmetric and reflective by definition. By the confluence, we can also prove its transitivity. See Appendix C.9.

To define a successful state, we introduce a new process constructor $\surd$, and extend the typing system to accept $\surd$, and extend the erasure to preserve $\surd$.

A process P is accepting if $P \equiv \surd | P'$ for some P'. This is formally defined in Appendix C.9.

We list the new formulation for all the criteria and state the theorem. For the motivation of each criterion, see [51]. Below, for the sake of readability, we omit Γ and Δ from the encoding.

COMPOSITIONALITY CRITERION *For every* k*-ary typing rule* R *in the typing system of* $\mathcal{L}_1$ *and every subset of names* N *there exists a* k*-ary context* $C_R^N(\__1, \ldots, \__k)$ *such that, for all* $\mathcal{D}_1, \ldots, \mathcal{D}_k$ *with* $\mathrm{FN}(\mathcal{D}_1, \ldots, \mathcal{D}_k) = N$*, it holds that* $[\![R(\mathcal{D}_1, \ldots, \mathcal{D}_k)]\!] = C_R^N([\![\mathcal{D}_1]\!], \ldots, [\![\mathcal{D}_k]\!])$. Note that the information given by derivation (typing) in $\mathcal{D}_1 :: P_1$ and $\mathcal{D}_2 :: P_2$ are essential.

NAME INVARIANCE CRITERION *For every typing derivation* $\mathcal{D} :: P$ *(*P *has derivation* $\mathcal{D}$*) and name substitution* σ*, it holds that if* σ *is injective, then* $[\![\mathcal{D}\sigma]\!] = [\![\mathcal{D}]\!]\sigma'$*; for every* $a \in \mathcal{N}$*, otherwise* $[\![\mathcal{D}\sigma]\!] \asymp_2 [\![\mathcal{D}]\!]\sigma'$ *where* σ' *is such that* $\varphi_{[\![]\!]}(\sigma(a)) = \sigma'(\varphi_{[\![]\!]}(a))$*. Here* $\varphi_{[\![]\!]}$ *is called the renaming policy and captures how* $[\![\cdot]\!]$ *translates channel names.*

OPERATIONAL CORRESPONDENCE CRITERION Let $\to_i$ denote the reduction relation of the system i.
(1) Completeness: *If* $\mathcal{D}_1 :: P_1$ *and* $P_1 \to_1^\star P_2$ *then there exists a* $\mathcal{D}_2 :: P_2$ *such that* $[\![\mathcal{D}_1]\!] \to_2^\star \asymp_2 [\![\mathcal{D}_2]\!]$.
(2) Soundness: *If* $[\![\mathcal{D}_1 :: P_1]\!] \to_2^\star Q_1$ *then there exists a* $\mathcal{D}_2 :: P_2$ *such that* $P_1 \to_1^\star P_2$ *and* $Q_1 \to_2^\star \asymp_2 [\![\mathcal{D}_2]\!]$.

DIVERGENCE REFLECTION CRITERION *If* $[\![\mathcal{D} :: P]\!] \to^\omega$ *then* $P \to^\omega$ where $\to^\omega$ means infinite reductions.

SUCCESS SENSITIVENESS CRITERION *If* $\mathcal{D} :: P$ *then* $P \Downarrow$ *if and only if* $[\![\mathcal{D}]\!] \Downarrow$ where $P \Downarrow$ means P can reach a successful state.
Using the above definition, we arrive at the following main theorem. See Appendix C.9 for the proofs.

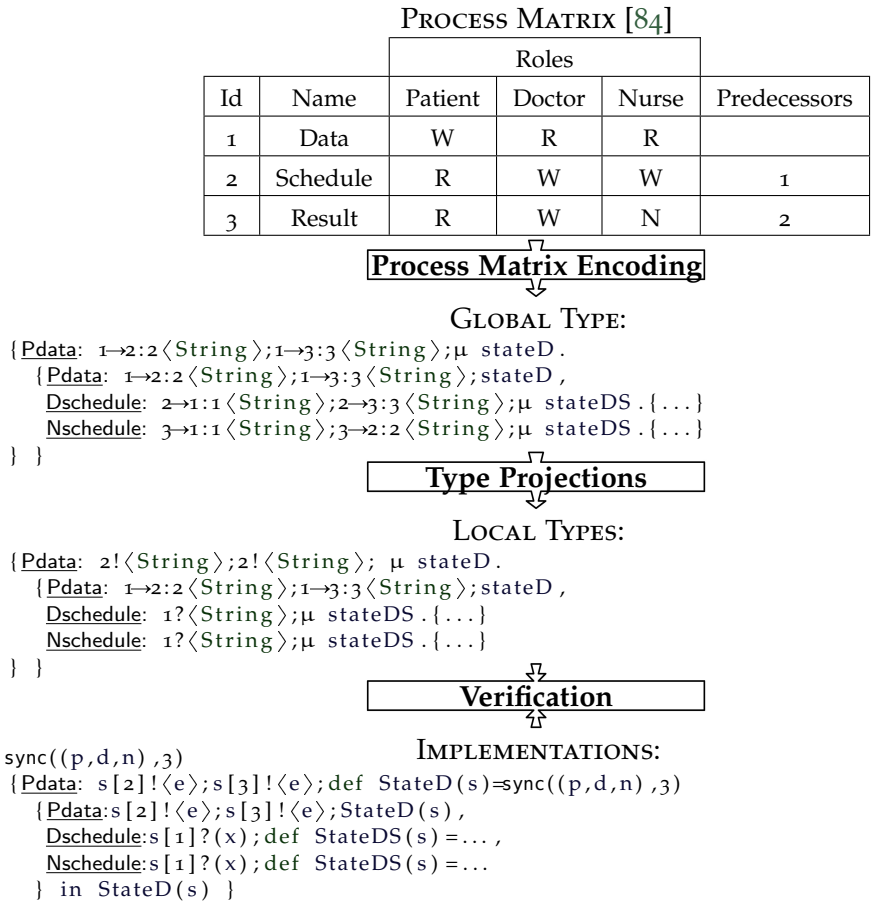
Theorem 4.4.6. *The erasure mapping satisfies all the encodability criteria.*

4.5 VERIFYING CPG DESCRIPTIONS

This section describes how symmetric sum types can verify implementation conformance to a CPG [115] described using the Process Matrix. The verification is performed by three steps in Fig. 50, as illustrated below.

PROCESS MATRIX. The Process Matrix representation consists of a table with one row for each action. Each row has a number of columns: The **Id** and **Name** columns

Figure 50 Steps in verifying a CPG description



are used to identify the action, and the **Predecessors** column holds the *Ids* of the actions the action depends on. Before an action can be executed its predecessors must have been executed. If all the predecessors of an action have been executed we say that the action is *executable*. Finally there is one column for each participant (called roles), where the content is either `R` meaning the participant can read the action-data but not execute it, `W` meaning the participant can execute the action and read its data or `N` meaning the participant cannot execute the action or read its data (see [84] for a more adequate description). The Process Matrix in Fig. 50 describes the scenario from the introduction, except that the patient automatically gives the data to both the doctor and the nurse, and the user can perform the actions multiple times (by an implicit recursion), until all the actions are executed.

PROCESS MATRIX ENCODING Any CPG in a Process Matrix can be encoded as a global type automatically. We explain this encoding by translating the above Process Matrix example. In the resulting type, the state is described by the set of actions that have been executed, leading to a finite but exponential number of states. The representation of each state (except the completed state) is a symmetric sum with one branch for each role that can execute each executable action. The content of each branch consists of the executing participant sending the created data to all other participants with read or write access, followed by the state where the executed action is added, and depending actions have been removed.

Parts of the global type is included in Fig. 50. Notice that the resulting type uses recursion: this is to describe an implicit recursion in the Process Matrix where the state reached after an action does not have to be a new state, but can be the same as the state before the execution of the action, or even from previous steps. This is the case for the above example if the data is sent, the appointment is scheduled, and then the data is resent. The resulting state would then be the state where only the data action has been executed, which is the same as the second state. The described method can be extended to translate any Process Matrix into a global type.

The conversion of CPGs from the Process Matrix, to session type allows the data to be exchanged directly between the participants, while the current implementations rely on a centralised database for the exchange. This means the translation offers a distributed implementation of the Process Matrix, which has not been known before. A formally defined symmetric global synchronisation primitive, together with its type discipline and encodability, offers a firm basis for such implementations.

PROJECTION AND VERIFICATION When we have created the global type expressing the CPG, a process implementing one of the participants can be verified to conform with the workflow, by projecting the global type to the local type of that participant, and typechecking the process against the local type. Parts of the local type and the process for the Patient are described in Fig. 50.

GENERALISATION We have now described how to use the multiparty session types extended with symmetric sum, to express CPGs formalised using the Process Matrix. We believe many other workflow frameworks (such as large parts of the BPMN) can be encoded as multiparty session types with symmetric sum, and this would allow the type-system to serve as a common representation, enabling interaction between different frameworks and implementing features (such as automatic user-interface generation) only for symmetric sum types, and apply it to all the encoded frameworks.

4.5.1 *Implementation*

We have created an `ascii` syntax for the asynchronous π-calculus with multiparty sessions and symmetric synchronisation called APIMS, and implemented a typechecker and an interpreter. This is to our knowledge the first prototype implementation of the π-calculus with multiparty sessions and multiparty session types. The implementation along with example programs can be found on the APIMS website [1].

The implementation extends the calculus with a `guisync` constructor to support user interaction via GUIs. The `guisync` is the result of extending the `sync` for user input. Each label has a set of typed arguments that must be given using the GUI before that choice is accepted, and the given arguments can be used by the process in that branch. This simple extension allows the processes to implement GUIs and the type system guarantees that the GUI for each participant will respect the protocol, hence the workflow. The mandatory labels ensure that the GUI must allow all the users (the people using the interface for each participant) to agree in each synchronisation, thus avoiding the GUIs causing a disagreement w.r.t. the theory of a symmetric synchronisation.

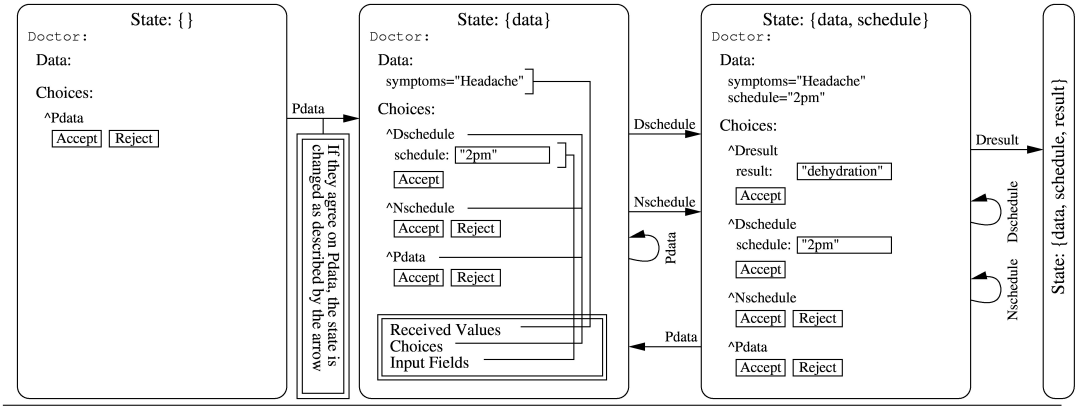
The GUI shows the received data, the choices offered by the process, input fields for the data needed for each choice, and buttons to accept/reject each choice. Fig. 51 shows three screen-shots, displaying the doctor's GUI for each state and how each choice affects the state. As soon as all the participants of a session accepts the same choice, the processes continue with the accepted branch. The GUI implementation for each participant can be created automatically from the Process Matrix.

The original implementation of the Process Matrix called *Online Consultant* by *Resultmaker* [84] is database based. This means that communication consists of the sender uploading information to the server, and all participants must query the server when using the information. Implementing the workflows using the π-calculus and session types not only gives the *Process Matrix* a formal semantics, but also allows an implementation where participants communicate their data as peer-to-peer. This offers more natural and robust realisation of the workflows, and relieves the system from the server bottleneck.

4.6 RELATED AND FUTURE WORK

There are existing studies on self/broadcast synchronisations [59, 101]. The symmetric sum proposed in the present paper is different because it allows all the participants to influence the choice equally and, to formulate this notion adequately, demands a session-based operational framework. Another difference is the use of the type discipline to control this complex synchronisation framework, which is not found in the foregoing work. Note that the type discipline allows multiparty

Figure 51 States and screenshots for the doctor GUI



progress and communication-safety for participants, which is not generally ensured in existing untyped self/broadcast synchronisation primitives. Our primitive and its type-checker are applicable not only to Process Matrix, but also multiparty synchronisations in general with strong safety guarantees.

The symmetric synchronisation is similar to the consensus in Weak Byzantine Agreement (WBA) [14, 40, 41, 81] which is a formalisation of the database commit problem. The similarity is that a number of processes need to end up with a common choice. In contrast to symmetric sum, WBA only has two possible choices (0 and 1). Not all participant has to initially accept the final decision, but if all processes agree initially, the result should be the initial preference. WBA is studied in an untyped settings on unreliable networks, with faulty processes (with arbitrary behaviour).

The symmetric sum is also similar to the symmetric choice $\Box$ in CSP and the mixed choice in the π-calculus [99]. The main difference is these preceding primitives are restricted to two party synchronisations. Our result is consistent with the non-encodability of the mixed-choice π-calculus in the separated choice π-calculus [99]: our erasure is defined on *typing derivations*, and cannot be made homomorphic on *processes*. For example, take $P = (\nu a)(P_1|P_2)$ where

$$P_1 = \overline{a}[2](s).\mathtt{sync}\{\underline{l1} : P_{11}, l2 : P_{12}\} \text{ and}$$
$$P_2 = a[2](s).\mathtt{sync}\{\underline{l1} : P_{21}, l3 : P_{23}\}.$$

This process shows that the erasure cannot be interpreted as an encoding from processes $[\![\cdot]\!]$ where $[\![P_1|P_2]\!] = [\![P_1]\!]|[\![P_2]\!]$, because the result of $[\![P_1]\!]$ depends on the context P_1 is in: the conductor inserted by the second step of the erasure depends on the type of a which depends on the other process. In the given context, the conductor must consider the labels $l1, l2$ and $l3$, and this could not be generated from $[\![P_1]\!]$ because P_1 does not contain any information about $l3$. As noted above, the symmetric sum and synchronisation construct differs from the mixed choice

and from the untyped asymmetric, directed sums whose encodability is studied in [90, 91], in that it is multi-party synchronisation for a fixed number of participants ensured by the underlying session type discipline.

Types for the multiparty interactions are studied in the *conversation calculus* [26] and *contracts* [28]. The former has choice behaviours where the channel-based communication is replaced by conversation environments allowing multiple participants, while the latter uses a process-based specification of protocols relying on internal and external choices, where conformance is formalised based on must preorder (so that we can ensure liveness). Our implementation crucially relies on the choreographic description based on global types: in particular, global types can offer a tractable, clear type-directed generation from the Process Matrices as described in Section 4.5.

As future work, we plan to extend our work with logical assertions based on [20] in order to describe and ensure the communicated data fulfil desired properties (for example, "the prescribed medicine doses are less than the lethal amount"). With the assertions, we can add arguments (state) to the recursive types, and conditions to the branches in a choice, so that it will lead to a more efficient generation from the Process Matrix.

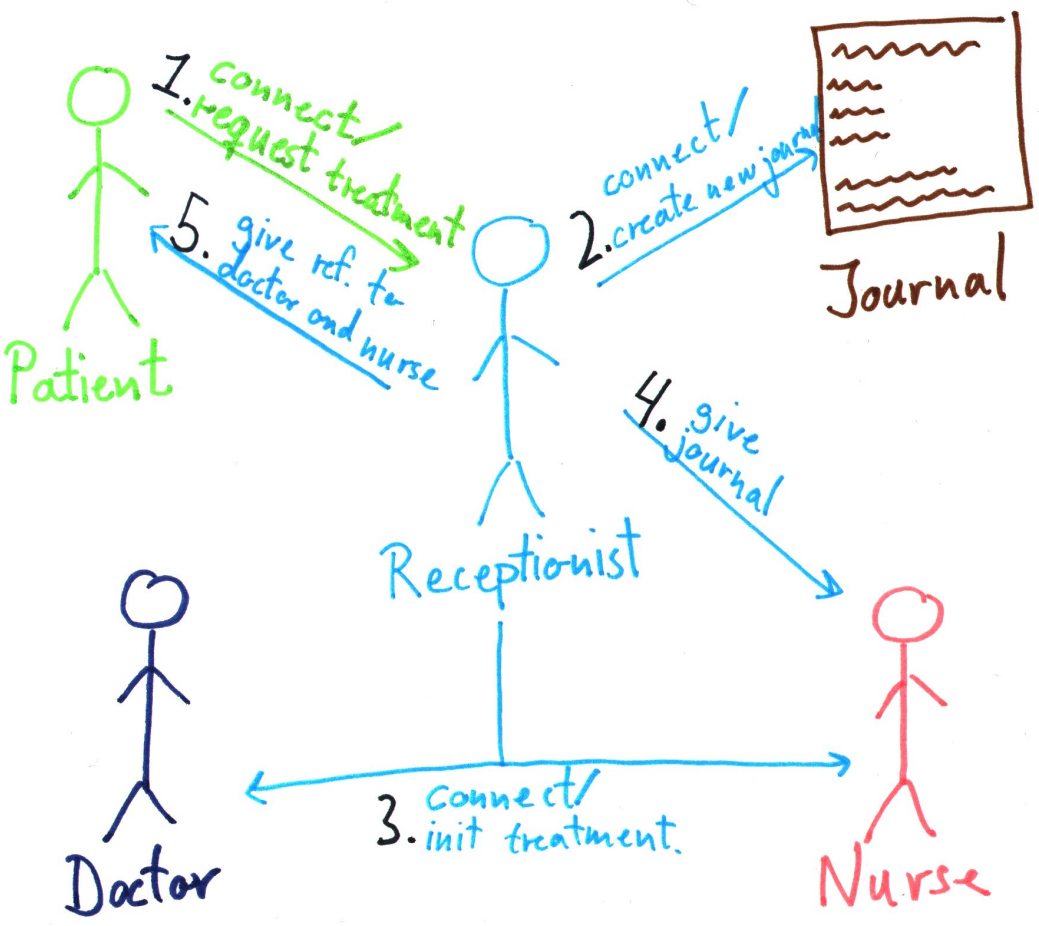
Acknowledgements

The first author is supported by the *TrustCare* project, funded by the Danish Strategic Research Agency, Grant #2106-07-0019. The last two authors are partially supported by EPSRC EP/F003757, EP/F002114, EP/G015635 and EP/G015481.

5

MULTIPARTY SYMMETRIC SUM TYPES WITH ASSERTIONS

AUTHORS:
LASSE NIELSEN - UNIVERSITY OF COPENHAGEN,
NOBUKO YOSHIDA - IMPERIAL COLLEGE LONDON AND
KOHEI HONDA - QUEEN MARY UNIVERSITY OF LONDON



This paper merges and refines two extensions of multiparty session types, adding assertions and the symmetric sum type constructor to the multiparty session type language. Assertions are used for increasing the expressiveness of types to ensure that the communicated values respect the asserted predicates as well as the type domain. Symmetric sum types represent a decision by collective agreement, and can be used to express cooperational workflows such as clinical practice guidelines in the typing system, ensuring that certain choices and actions are made according to the workflow. The combination of the two extensions to multiparty session types allows the workflows represented to be more expressive and compact. This paper formalises an extension of the asynchronous π-calculus with multiparty sessions and the typing system including both assertions and symmetric sum types. We show that the properties of multiparty session types are preserved by this extension, and investigate what examples can be expressed by the extension and how efficiently. Finally we present an implementation of the full framework, where the assertion language has been restricted to a decidable fragment, and implementations of real workflows that can be verified and executed by the implemented tool.

5.1 INTRODUCTION

Multiparty session types [61] can be used to define protocols for interactions in a group of participants, and verify that π-calculus processes follow the specified protocol. Design-by-Contract (DbC) [20] extends the multiparty session types with assertions, which elaborates type signatures though logical predicates. This can be used to restrict the values that are communicated and choices that are made.

Symmetric sum types [96] is an extension of the multiparty session types that can type nondeterministic orchestrational choice behaviors. This can be used to represent workflows such as clinical practice guidelines (CPGs) [115] as types, such that implementations can be verified to comply with the represented workflow by type checking. CPGs are detailed descriptions of medical treatment procedures, practised globally with local variations, in order to treat specific medical disorders. CPGs are an example of social interactions, which include workflow models and various cooperation models: its richness stems from the diverse collaborative patterns human organisations can exhibit. One such pattern, which plays a prominent role in CPGs, is *symmetric synchronisation* where all the participants are equal in the decision-making, i.e. the participants collectively decide on one of the possible choices.

This paper merges the two extensions to obtain a framework with both symmetric sum types and assertions. The motivation for combining these extensions is that assertions are directly usefull in the context of CPGs, but also that workflows in general can be represented more compactly using assertions.

To illustrate how assertions allow more compact representation of workflows, we have described a typical CPG workflow in Fig. 52 using the Buisness Process Modelling Notation (BPMN). The described workflow is activated, when a patient is admitted. First two tests are executed, possibly in parallel. Then, depending on the result of the tests, either the patient is discharged directly, or the patient is treated before discharging. In this workflow the treatment consists of administering a drug to the patient. The workflow is ended, when the patient is discharged. The described workflow is a standard paradigm in CPGs, that is, first a set of tests

Figure 52 Typical CPG workflow in BPMN

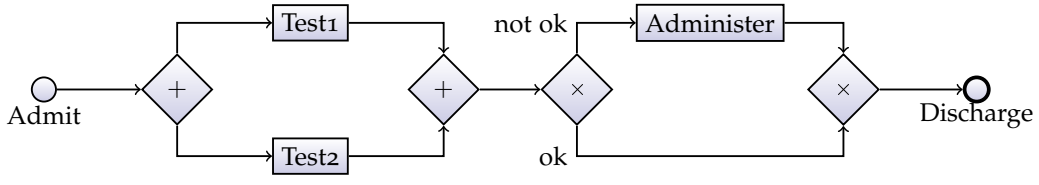


Figure 53 Session type representation of workflow

```
{ Test1 :
    3→1:1⟨Bool⟩;   // The result of
        test1
    3→2:2⟨Bool⟩;   // The result of
        test1
    { Test2 :
        3→1:1⟨Bool⟩; // The result of
            test2
        3→2:2⟨Bool⟩; // The result of
            test2
        TREAT
    },
    Test2 :
    3→1:1⟨Bool⟩;   // The result of
        test2
    3→2:2⟨Bool⟩;   // The result of
        test2
    { Test1 :
        3→1:1⟨Bool⟩; // The result of
            test1
        3→2:2⟨Bool⟩; // The result of
            test1
        TREAT
    }
}
```

a) Type without assertions

```
μ  workflow⟨test1:Bool=false,test2:Bool=false⟩
     .
{ Test1 [[not test1]]:
    3→1:1⟨Bool⟩; // The result of test1
    3→2:2⟨Bool⟩; // The result of test1
     workflow⟨true,test2⟩,
    Test2 [[not test2]]:
    3→1:1⟨Bool⟩; // The result of test2
    3→2:2⟨Bool⟩; // The result of test2
     workflow⟨test1,true⟩,
    Diagnose [[test1 and test2]]:
    TREAT'
}
```

b) Type using assertions

are performed, and depending on the results, either more tests are performed, the patient is discharged or a treatment is executed.

If we try to describe the workflow as a multiparty session type with symmetric

Figure 54 Session type representation of workflow using assertions

```
μ  workflow⟨test1:Bool=false,test2:Bool=false,administer:Bool=false,
            result1:Bool=false,result2:Bool=false⟩.
{ Test1 [[not test1]]:
    3→1:1⟨Bool⟩ as x;          // The result of test1
    3→2:2⟨Bool⟩ as y[[x=y]]; // The result of test1
     workflow⟨true,test2,administer,x,result2⟩,
    Test2 [[not test2]]:
    3→1:1⟨Bool⟩ as x;          // The result of test2
    3→2:2⟨Bool⟩ as y[[x=y]]; // The result of test2
     workflow⟨test1,true,administer,result1,x⟩,
    Administer [[test1 and test2 and not administer and not (result1 and result2)]]:
     workflow⟨test1,test2,true,result1,result2⟩,
    Discharge [[test1 and test2 and (result1 or result2 or administer)]]:
    end
}
```

sum without using assertions, we can use the syntax from the original symmetric sum type paper [96], where $\{l : G_l\}_{l \in L;M}$ represents a choice by common agreement where the possible choices are the labels in $L \cup M$ and G_l describes how to procede for the choice l. With this syntax, we can represent the workflow from Fig. 52 as the type in Fig. 53a. The type states that first either Test1 or Test2 is executed. When Test1 is executed, the result is sent from participant 3 to participant 1 and 2, this could for example be the nurse informing the patient and the doctor of the result. If Test1 was executed, then Test2 is executed (again the result is communicated), and then the workflow proceeds as TREAT which represents the type for evaluating the test results and treating the patient accordingly before discharging. If Test2 is executed first, then Test1 is executed before continuing with TREAT. It is noticable, how each permutation of the tests must be allowed explicitly. This is not so bad in the given example, as the number of tests is 2, but as the number of tests n increases, the number of permutation explodes, because there are $n!$ permutations.

Using the approach from the original assertions paper [20], we can extend the syntax of the symmetric sum types to $\{\{A_l\}\, l : P_l\}_{l \in L;M}$, where A_l is the assertion on the choice l meaning that l can only be chosen if A_l is valid. Using the extended syntax, the workflow can be described by the type in Fig. 53b. The workflow is described by a recursive type. This may be surprising, as the original workflow has no recursion, but the idea is that the recursion has a state (test1 and test2 describing if the respective actions have been executed), which is used to decide what actions can be executed. After executing an action, the recursive type is used with an updated state. In the state where both tests have been executed, the Diagnose action can be used to continue with the remaining workflow represented as TREAT'.

Since the compact representation uses assertions, it makes sense to consider if the assertions are useful for other aspects of the CPG workflows, and this seems like a perfect fit. Assertions can for example be used to ensure that the prescribed doses of medicine are below the lethal limit. In the example workflow the assertions can be used to ensure, that the result sent to the patient and the doctor after a test is the same, and the results of the tests can for instance be used to indicate if it makes sense to administer the medicine or discharge the patient directly. This is used in the type in Fig. 54 representing the full workflow, where the results communicated in each test is used to decide if the Administer and Discharge actions can be selected.

In the remaining sections, we will formalise the framework with both symmetric sum and assertions, prove that the properties of the original multiparty session types are preserved, and show a working implementation of the framework and real world examples represented and verified using the implementation.

Appendix D includes the omitted definitions and proofs, but the paper can be read independently.

Figure 55 The process language

$P ::= \mathsf{sync}_{\tilde{s},n}\{\{A_l\}\, l : P_l\}_{l\in L}$	synchronisation	$\mid s \lhd l; P$	label selection
$\mid 0$	inaction	$\mid s \rhd \{l : P_l\}_{l\in L}$	label branching
$\mid \overline{a}[2..n](\tilde{s}).P$	session request	$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$\mid a[p](\tilde{s}).P$	session accept	$\mid P \mid Q$	parallel
$\mid s!\langle \tilde{e} \rangle; P$	value sending	$\mid (\nu n)P$	restriction
$\mid s?(\tilde{x}); P$	value reception	$\mid \mathsf{def}\ D\ \mathsf{in}\ P$	recursion
$\mid s!\langle\!\langle \tilde{s} \rangle\!\rangle; P$	delegation	$\mid X\langle \tilde{e}\tilde{s} \rangle$	process call
$\mid s?(\!(\tilde{s})\!); P$	reception	$\mid s : \tilde{h}$	message queue
$D ::= \{X_i\langle \tilde{x}_i \rangle(\tilde{s}_i) = P_i\}_{i\in I}$	declarations	$v ::= a \mid \mathsf{true} \mid \mathsf{false}$	values
$e ::= v \mid x \mid e\ \mathsf{and}\ e' \mid \ldots$	expressions	$h ::= l \mid \tilde{v} \mid \tilde{s}$	messages
$A ::= e$	assertions		

5.2 THE PROCESS LANGUAGE

This section introduces the syntax (Fig. 55) of the asynchronous multiparty session π-calculus [61] with the new sync primitive, and the judgement $P \to P'$ (Fig. 56,

Figure 56 The reduction rules

$$\frac{}{\overline{a}[2..n](\tilde{s}).P_1 \mid a[2](\tilde{s}).P_2 \mid \ldots \mid a[n](\tilde{s}).P_n \to (\nu\tilde{s})(P_1 \mid P_2 \mid \ldots \mid P_n \mid s_1 : \varnothing \mid \ldots \mid s_m : \varnothing)}\ [\textsc{Link}]$$

$$\frac{\tilde{e} \downarrow \tilde{v}}{s!\langle \tilde{e} \rangle; P \mid s : \tilde{h} \to P \mid s : \tilde{h} \cdot \tilde{v}}\ [\textsc{Send}] \qquad \frac{}{s?(\tilde{x}); P \mid s : \tilde{v} \cdot \tilde{h} \to P[\tilde{v}/\tilde{x}] \mid s : \tilde{h}}\ [\textsc{Recv}]$$

$$\frac{}{s \lhd l; P \mid s : \tilde{h} \to P \mid s : \tilde{h} \cdot l}\ [\textsc{Label}] \qquad \frac{j \in I}{s \rhd \{l_i : P_i\}_{i\in I} \mid s : l_j \cdot \tilde{h} \to P_j \mid s : \tilde{h}}\ [\textsc{Branch}]$$

$$\frac{}{s!\langle\!\langle \tilde{t} \rangle\!\rangle; P \mid s : \tilde{h} \to P \mid s : \tilde{h} \cdot \tilde{t}}\ [\textsc{Deleg}] \qquad \frac{}{s?(\!(\tilde{t})\!); P \mid s : \tilde{t} \cdot \tilde{h} \to P \mid s : \tilde{h}}\ [\textsc{SRec}]$$

$$\frac{e \downarrow \mathsf{true}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to P}\ [\textsc{IfT}] \qquad \frac{e \downarrow \mathsf{false}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to Q}\ [\textsc{IfF}]$$

$$\frac{\tilde{e} \downarrow \tilde{v} \quad X\langle \tilde{x}\tilde{s} \rangle = P \in D}{\mathsf{def}\ D\ \mathsf{in}\ X\langle \tilde{e}\tilde{s} \rangle \mid Q \to \mathsf{def}\ D\ \mathsf{in}\ P[\tilde{v}/\tilde{x}] \mid Q}\ [\textsc{Def}] \qquad \frac{P \to P'}{(\nu n)P \to (\nu n)P'}\ [\textsc{Scop}]$$

$$\frac{P \to P'}{P \mid Q \to P' \mid Q}\ [\textsc{Par}] \qquad \frac{P \to P'}{\mathsf{def}\ D\ \mathsf{in}\ P \to \mathsf{def}\ D\ \mathsf{in}\ P'}\ [\textsc{Defin}] \qquad \frac{P \equiv P' \quad P \to Q \quad Q \equiv Q'}{P' \to Q'}\ [\textsc{Str}]$$

$$\frac{h \in \bigcap_{i=1}^{n} L_i \quad A_{1h} \downarrow \mathsf{true} \quad \cdots \quad A_{nh} \downarrow \mathsf{true}}{\mathsf{sync}_{\tilde{s},n}\{\{A_{1l}\}\, l : P_{1l}\}_{l\in L_1} \mid \ldots \mid \mathsf{sync}_{\tilde{s},n}\{\{A_{nl}\}\, l : P_{nl}\}_{l\in L_n} \to P_{1h} \mid \ldots \mid P_{nh}}\ [\textsc{Sync}]$$

where $e \downarrow v$ denotes the evaluation of the expression e to the value v) describing the small-step semantics for processes. The syntax defines the values: $\{v, w, \ldots\}$, expressions: $\{e, e', \ldots\}$, assertions $\{A, B, \ldots\}$ and processes: $\{P, Q, \ldots\}$ from the sets of channel names: $\{a, b, \ldots\}$, value variables: $\{x, y, \ldots\}$, session channels: $\{s, t, \ldots\}$, labels: $\{l, m, \ldots\}$ and process variables: $\{X, Y, \ldots\}$.

It is a noticeable difference from the original assertion paper [20], that the assertions are not included in the processes. Only the synchronization construct explicitly includes the assertions, and this is because the assertions are a vital part of the synchronisation as they express what choices the process accepts, and

Figure 57 Implementation of the participants in the workflow from Fig. 52 and Fig. 54

```
P_P = // Patient
ā[2..3](p,d,n).
def X⟨t1:Bool,t2:Bool,adm:Bool,
      r1:Bool,r2:Bool⟩
     ((p,d,n): workflow↾1⟨t1,t2,adm,
                          r1,r2⟩)=
  sync((d,s,r),3)
  { Test1 [[not t1]]:
    p?(result);
    X⟨true,t2,adm,result,r2⟩((p,d,m)),
    Test2 [[not t2]]:
    p?(result);
    X⟨t1,true,adm,r1,result⟩((p,d,m)),
     Administer [[t1 and t2 and not adm
                  and not (r1 and r2)]]:
    X⟨t1,t2,true,r1,r2⟩,
     Discharge [[t1 and t2 and
                (r1 or r2 or adm)]]:
    end
  }
in X⟨false,false,false,false,false⟩((p,d,n))
```

```
P_D = // Doctor
a[2](p,d,n).
def X⟨t1:Bool,t2:Bool,adm:Bool,
      r1:Bool,r2:Bool⟩
     ((p,d,n): workflow↾1⟨t1,t2,adm,
                          r1,r2⟩)=
  sync((d,s,r),3)
  { Test1 [[not t1]]:
    d?(result);
    X⟨true,t2,adm,result,r2⟩((p,d,m)),
    Test2 [[not t2]]:
    d?(result);
    X⟨t1,true,adm,r1,result⟩((p,d,m)),
     Administer [[t1 and t2 and not adm
                  and not (r1 and r2):
    X⟨t1,t2,true,r1,r2⟩,
     Discharge [[t1 and t2 and
                (r1 or r2 or adm)]]:
    end
  }
in X⟨false,false,false,false,false⟩((p,d,n))
```

```
P_N = // Nurse
a[3](p,d,n).
def X⟨t1:Bool,t2:Bool,adm:Bool,r1:Bool,r2:Bool⟩
     ((p,d,n): workflow↾1⟨t1,t2,adm,r1,r2⟩)=
  sync((d,s,r),3)
  { Test1 [[not t1]]:
    p!∉Result⟩; d!∉Result⟩; X⟨true,t2,adm∉Result,r2⟩((p,d,m)),
    Test2 [[not t2]]:
    p!∉Result⟩; d!∉Result⟩; X⟨t1,true,adm,r1∉Result⟩((p,d,m)),
     Administer [[t1 and t2 and not adm and not (r1 and r2):
    X⟨t1,t2,true,r1,r2⟩,
     Discharge [[t1 and t2 and (r1 or r2 or adm)]]:
    end
  }
in X⟨false,false,false,false,false⟩((p,d,n))
```

the used assertions for optional branches does not have to be equivalent to the assertions used in the type. The reason why the original assertion paper includes the assertions in the processes for sending and receiving messages and branches is, that they are used to describe a semantics where sending or receiving a message that does not meet the assertions result in an error, and this can be used to prove a 'well typed processes do not go wrong' result. The approach used in this paper is to describe an unsafe semantics, meaning that messages that does not meet the assertions can be communicated without detection. This means that we cannot prove the same result as the original assertion paper, but we will in stead use the approach of the original multiparty session type paper [61] to prove that each step of a well type process, can be matched by a step by the type-environment such that the result process is well-typed in the result environment. This also means that the communicated messages meets the assertions, as this is required by the environment steps.

Session request ($\overline{a}[2..n](\tilde{s}).P$) initiates a session with channels $\tilde{s}$ (where $\tilde{s}$ denotes a vector $s_1 \dots s_n$) over the public channel a with the other $n-1$ participants of shape $a[p](\tilde{s}).Q_p$ for p from 2 to n ([Link] in Fig. 56). Asynchronous communication in an established session is performed by sending and receiving values ([Send,Recv]), transferring a session using session delegation and reception ([Deleg,SRec]), and label selection and branching ([Label,Branch]), where the branching process offers a number of labels and the selecting process chooses one of them.

The new $\mathtt{sync}_{\tilde{s},n}\{\{A_l\}l : P_l\}_{l\in L}$ constructor is interpreted as the process participating in a plenum decision between all the n processes in the session $\tilde{s}$ reaching a common decision h from L, which all the processes accept since the assertions evaluate to true. Afterwards each process p proceeds as described in P_{ph}. In [Sync] in Fig. 56, h in the premise denotes the common label.

In [Sync], the processes cannot perform the synchronisation if they do not accept some common label, in which case the processes will be stuck. We also need to know how many participants are in the session in order to know when the synchronisation can step; otherwise the processes will be stuck, or some processes will be left behind. The typing system introduced in the next section ensures that `sync` satisfies such conditions.

WORKFLOW EXAMPLE (1): PROCESSES We give implementations of all three participants in the workflow from the introduction in Fig. 57.

5.3 THE TYPE LANGUAGE

We start by defining the global types G in Fig. 58, which specifies global session protocols between the participants. Except for the symmetric sum type, the syntax is from [20]. The type $p \to p' : k\langle U\rangle \ \mathtt{as}\ x\{A\}.G'$ expresses that participant p sends

Figure 58 The Domains used for Global and Local types

(Global Types)

$$\begin{aligned} G ::= {} & p \to p' : k\langle S\rangle \texttt{ as } x\{A\}.G' \\ | {} & p \to p' : k\langle U\rangle.G' \\ | {} & p \to p' : k\{\{A_i\}\, l_i : G_i\}_{i\in I} \\ | {} & \mu t\langle \tilde{x}\rangle(\tilde{e}).G \\ | {} & t\langle \tilde{e}\rangle \\ | {} & \mathsf{end} \\ | {} & \{\{A_l\}\, l : G_l\}_{l\in L;M} \end{aligned}$$

(Local Types)

$$\begin{aligned} T ::= {} & k!\langle S\rangle \texttt{ as } x\{A\};T \\ | {} & k?\langle S\rangle \texttt{ as } x\{A\};T \\ | {} & k!\langle U\rangle;T \mid k?\langle U\rangle;T \\ | {} & k \oplus \{\{A_l\}\, l : T_l\}_{l\in L} \\ | {} & k \,\&\, \{\{A_l\}\, l : T_l\}_{l\in L} \\ | {} & \mu t\langle \tilde{x}\rangle(\tilde{e}).T \mid t\langle \tilde{e}\rangle \mid \mathsf{end} \\ | {} & \{\{A_l\}\, l : T_l\}_{l\in L;M} \\ | {} & \forall x : S\{A\}.T \end{aligned}$$

(Message Types)

$$U ::= \tilde{S} \mid T@(p, m, n)$$

(Simple Types)

$$S ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots \mid \langle G\rangle$$

(Environments)

$$\Gamma ::= \varnothing \mid \Gamma, u : \langle G\rangle \mid \Gamma, X : \tilde{S}\tilde{T}$$

$$\Delta ::= \varnothing \mid \Delta, \tilde{s} : T@(p, n)$$

$$\Theta ::= A$$

a message of type U along channel k to p' and then interactions described in G' take place. The $\texttt{as } x\{A\}$ parts binds occurances of x in G and A to the value communicated, and states that the value must respect the predicate A. The type $p \to p' : k\{\{A_i\}\, l_i : G_i\}_{i\in I}$ expresses that p sends one of the labels l_i to p'. If l_j is sent, then the predicate A_j must be fulfilled, and the interactions described in G_j take place. Type $\mu t\langle \tilde{x}\rangle.G$ is a recursive type where $\tilde{x}$ is the state, assuming type variables $(t, t', \ldots)$ are guarded in the standard way. We assume that G in the grammar of sorts is closed, i.e., without free type or assertion variables. Type end represents the session termination.

The sum type $\{\{A_l\}\, l : G_l\}_{l\in L;M}$ represents a synchronisation where the labels are taken from the set $L \cup M$. The labels in L are optional, but the labels in M are mandatory and must be accepted by all the participants. If the predicate A_l is false, the label is ignored, and must be rejected. The mandatory labels will be underlined to distinguish them from the optional labels (e.g. $\{l : G_l\}_{l\in\{l1\};\{l2\}} = \{l1 : G_{l1}, \underline{l2} : G_{l2}\}$).

The local types T are defined in Fig. 58. They describe the communication performed by a single process. Therefore the "from process to process on channel" syntax is simply changed to sending or receiving on a channel. Thus the sending type is $k!\langle U\rangle \texttt{ as } x\{A\};T$ and represents sending a message of type U on channel k respecting the predicate A, followed by the communication described by T. The type of receiving is $k?\langle U\rangle \texttt{ as } x\{A\};T$, the type of selecting is $k \oplus \{\{A_l\}\, l : T_l\}_{l\in L}$ and the type of branching is $k \,\&\, \{\{A_l\}\, l : T_l\}_{l\in L}$. The difference from the original assertion paper is that the symmetric sum type constructor $\{\{A_l\}\, l : T_l\}_{l\in L;M}$ is added where L, M satisfies the conditions similar to those of a global sum type, and we have introduced a $\forall x\{A\}.T$ constructor. This is to capture the local type of

Figure 59 Projection from global to local types ($\upharpoonright$)

$$
\begin{array}{rcl}
(p_0 \to p_1 : k\langle S\rangle \text{ as } x\{A\}.G')\upharpoonright p & = & \begin{cases} m!\langle S\rangle \text{ as } x\{A\};(G'\upharpoonright p) & \text{if } p = p_0 \text{ and } p \neq p_1 \\ m?\langle S\rangle \text{ as } x\{A\};(G'\upharpoonright p) & \text{if } p = p_1 \text{ and } p \neq p_0 \\ \forall x : S\{A\}.G'\upharpoonright p & \text{if } p \neq p_0 \text{ and } p \neq p_1 \end{cases} \\
(p_0 \to p_1 : k\langle U\rangle.G')\upharpoonright p & = & \begin{cases} m!\langle U\rangle;(G'\upharpoonright p) & \text{if } p = p_0 \text{ and } p \neq p_1 \\ m?\langle U\rangle;(G'\upharpoonright p) & \text{if } p = p_1 \text{ and } p \neq p_0 \\ G'\upharpoonright p & \text{if } p \neq p_0 \text{ and } p \neq p_1 \end{cases} \\
(p_0 \to p_1 : k\{\{A_j\}\, l_j : G_j\}_{j\in J})\upharpoonright p & = & \begin{cases} k \oplus \{\{A_j\}\, l_j : (G_j\upharpoonright p)\}_{j\in J} & \text{if } p = p_0 \neq p_1 \\ k \,\&\, \{\{A_j\}\, l_j : (G_j\upharpoonright p)\}_{j\in J} & \text{if } p = p_1 \neq p_0 \\ \forall\_\left\{\bigvee_{j\in J} A_j\right\}.\max_{\leqslant_{sub}} & \text{if } p_1 \neq p \neq p_2 \\ \quad \{T' \mid \forall j \in J.T' \leqslant_{sub} (G_j\upharpoonright p)\} & \end{cases} \\
(\{\{A_l\}\, l : G_l\}_{l\in L;L'})\upharpoonright p & = & \{\{A_l\}\, l : (G_l\upharpoonright p)\}_{l\in L;L'} \\
(\mu t\langle\tilde{x}\rangle(\tilde{e}).G)\upharpoonright p & = & \mu t\langle\tilde{x}\rangle(\tilde{e}).(G\upharpoonright p) \\
(t\langle\tilde{e}\rangle)\upharpoonright p & = & t\langle\tilde{e}\rangle \\
(\text{end})\upharpoonright p & = & \text{end}
\end{array}
$$

message parsing for a participant that is neither the sender or the receiver in a more intuitive way than the original assertion paper. In this case the local type should allow only the behaviour that is valid for all possible messages, and therefore the local type is represented by a forall construct.

The message type $T@(p, m, n)$ is used for delegation. It describes an open session, and includes information about the participant number p, the number of session channels m, and the number of participants n in the session together with a local type T describing the remaining communication.

Finally we define the global environment Γ containing the global types for shared channels u, and process variables X, the local type environment Δ containing the remaining session communication in Fig. 58, where $\tilde{s} : T@(p, n)$ means $\tilde{s}$ is an open session with n participants, where T describes the remaining communication for participant p, and the assertion environment Θ which holds the current assertions.

The *projection* $G \upharpoonright p$ of a global type G for a participant p generates the local type for the participant in an intuitive way. Projection is defined in Fig. 59. The differences from the definition in [61] is that the assertions are preserved, the communication between a second and third party is not ignored, but results in a forall construct holding the message assertion, and we have added a case for the symmetric sum type from [96] extended with assertions $(\{\{A_l\}\, l : G_l\}_{l\in L;M})\upharpoonright p = \{\{A_l\}\, l : (G_l\upharpoonright p)\}_{l\in L;M}$.

A global type G is coherent [61] if and only if the projection $G \upharpoonright p$ is defined for all

Figure 60 Selected typing rules

$$\text{[Sync]}\quad \frac{\begin{array}{c}\forall l \in L'' : \Theta \wedge A_l; \Gamma \vdash P_l \rhd \Delta, \tilde{s} : T_l@(p,n) \quad L'' \subseteq L \cup L' \\ \forall l \in L \backslash L'' :\vdash \Theta \Rightarrow \neg B_l \quad \forall l \in L'' :\vdash \Theta \Rightarrow (A_l \Rightarrow B_l) \\ \forall l \in L :\vdash \Theta \Rightarrow (B_l \Rightarrow A_l) \quad \vdash \Theta \Rightarrow \bigvee_{l\in L} B_l\end{array}}{\Theta; \Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{\{A_l\}\, l : P_l\}_{l\in L''} \rhd \Delta, \tilde{s} : \{\{B_l\}\, l : T_l\}_{l\in L;L'}@(p,n)}$$

$$\text{[Mcast]}\quad \frac{\Gamma \vdash a : \langle G\rangle \qquad |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \qquad \Theta; \Gamma \vdash P \rhd \Delta, \tilde{s} : (G\upharpoonright 1)@(1,n) \qquad n = \mathsf{max}(\mathsf{pid}(G))}{\Theta; \Gamma \vdash \overline{a}[2..n](\tilde{s}).P \rhd \Delta}\ (\mathsf{fv}(G\upharpoonright 1) = \varnothing)$$

$$\text{[Macc]}\quad \frac{\Gamma \vdash a : \langle G\rangle \qquad |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \qquad \Theta; \Gamma \vdash P \rhd \Delta, \tilde{s} : (G\upharpoonright p)@(p,n) \qquad n = \mathsf{max}(\mathsf{pid}(G))}{\Theta; \Gamma \vdash a[p](\tilde{s}).P \rhd \Delta}\ (\mathsf{fv}(G\upharpoonright p) = \varnothing)$$

$$\text{[SendA]}\quad \frac{\Gamma \vdash e : S \quad \Theta; \Gamma \vdash P \rhd \Delta, s : T[e/x]@(p,n) \quad \vdash \Theta \Rightarrow A[e/x]}{\Theta; \Gamma \vdash s_k!\langle e\rangle; P \rhd \Delta, \tilde{s} : k!\langle S\rangle \text{ as } x\{A\}; T@(p,n)}$$

$$\text{[RcvA]}\quad \frac{\Theta \wedge A; \Gamma, x : S \vdash P \rhd \Delta, \tilde{s} : T@(p,n)}{\Theta; \Gamma \vdash s_k?(x); P \rhd \Delta, \tilde{s} : k?\langle S\rangle \text{ as } x\{A\}; T@(p,n)}\ (x \notin \mathsf{fv}(\Theta) \cup \mathsf{fv}(\Delta))$$

$$\text{[Sel]}\quad \frac{\Theta; \Gamma \vdash P \rhd \Delta, \tilde{s} : T@(p,n) \quad h \in L \quad \vdash \Theta \Rightarrow A_h}{\Theta; \Gamma \vdash s_k \lhd h; P \rhd \Delta, \tilde{s} : k \oplus \{\{A_l\}\, l : T_l\}_{l\in L}@(p,n)}$$

$$\text{[Branch]}\quad \frac{\forall l \in L : \Theta \wedge A_l; \Gamma \vdash P_l \rhd \Delta, \tilde{s} : T_l@(p,n)}{\Theta; \Gamma \vdash s_k \rhd \{l : P_l\}_{l\in L} \rhd \Delta, \tilde{s} : k\&\{\{A_l\}\, l : T_l\}_{l\in L}@(p,n)}$$

$$\text{[Conc]}\quad \frac{\Theta; \Gamma \vdash P \rhd \Delta \quad \Theta; \Gamma \vdash Q \rhd \Delta'}{\Theta; \Gamma \vdash P|Q \rhd \Delta \circ \Delta'}\ (\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \varnothing)$$

$$\text{[Subs]}\quad \frac{\Theta; \Gamma \vdash P \rhd \Delta \quad \vdash \Theta \Rightarrow \Delta \leqslant \Delta'}{\Theta; \Gamma \vdash P \rhd \Delta'}$$

participants, and G does not allow racing conditions (linearity). We only consider coherent global types.

JUDGEMENT The typing judgement extends the original one [20] with symmetric sum types. The judgement $\Theta; \Gamma \vdash P \rhd \Delta$ states that assuming Θ the process P in the environment Γ performs exactly the session communication described in Δ.

The main rules are included in Fig. 60. The local types now carry information about the number of participants n and channels m. The number of participants and channels is determined at the session initialisation in the rules [Mcast] and

Figure 61 Local type for the patient

```
G↾1 = // Local type for Patient
μ  workflow⟨test1:Bool=false, test2:Bool=false,
     administer:Bool=false,
           result1:Bool=false, result2:Bool=false⟩.
{ Test1 [[not test1]]:
  1?⟨Sdata⟩ as x;
  forall y[[x=y]];
   workflow⟨true,test2,administer,x,result2⟩,
  Test2 [[not test2]]:
  1?⟨Sdata⟩ as x;
  forall y[[x=y]];
   workflow⟨test1,true,administer,result1,x⟩,
  Administer [[test1 and test2 and not administer and
               not (result1 and result2)]]:
   workflow⟨test1,test2,true,result1,result2⟩,
  Discharge [[test1 and test2 and
              (result1 or result2 or administer)]]:
  end
}
```

[Macc], where $\mathsf{sid}(G)$ denotes channels that appear in G and $\mathsf{pid}(G)$ denotes the participants that appear in G. The rule [Sync] checks that the synchronisation uses the correct number of participants, the accepted branches includes the mandatory ones exactly (if and only if) when their predicate is fulfilled, and does not accept (only if) the optional ones with a false predicate. It is checked that there is always an active mandatory option, and finally that each accepted branch is typed with the correct communication.

Since the process is reduced by each rule-application, the typability question $\Theta; \Gamma \vdash P \triangleright \Delta$ is decidable when $\vdash A$ is decidable.

HEALTHCARE COOPERATION (2): TYPES The global type describing the workflow from Fig. 52 was deduced in the introduction, and given in Fig. 54. The local type describing the behavior of each participant can be found by projection, and the local type describing the behavior of the patient is given in Fig. 61. Using the global type and its projections we can now typecheck the processes.

Proposition 5.3.1. $a : \langle G \rangle \vdash P_P \mid P_D \mid P_N \triangleright \varnothing$.

We end this section by proving *subject reduction*, from which we can derive soundness, communication safety and single session progress [61, § 5] as corollaries. The formulation uses the extension of the typing to runtime processes ($\Theta; \Gamma \vdash P \triangleright_{\tilde{t}} \Delta$), which corresponds to the presented typing on processes without open sessions, but also accept processes with open sessions. This is obtained by joining compatible session environments (Δ, Δ') using the $\Delta \circ \Delta'$ operation to a single environment expressing the communication in both Δ and Δ'. Below $\Delta \to^{0/1} \Delta'$

denotes zero or one step using the type reduction [61], which represents the communication between dual local types. For this we use forall contexts, defined by

$$C^x ::= [\,] \mid \forall y : S\,\{A\}\,.C^x$$

where $y \neq x$. For instance, a reduction between input and output types is defined as:

$$\frac{\vdash v : S \quad A[v/x] \downarrow \mathsf{true}}{\left\{\begin{array}{l} C_1^x[k!\langle S\rangle\ \mathtt{as}\ x\,\{A\}\,;T_1@(p_1,n)], \\ C_2^x[k?\langle S\rangle\ \mathtt{as}\ x\,\{A\}\,;T_2@(p_2,n)], \\ C_3^x[\forall x : S\,\{A\}\,.T_3@(p_2,n)], \\ \ldots, \\ C_n^x[\forall x : S\,\{A\}\,.T_n@(p_n,n)] \end{array}\right\} \to \{T_p[v/x]@(p,n)\}_{p=1}^{n}}$$

We extend it to the symmetric sum as:

$$\frac{A_1 \downarrow \mathsf{true} \quad \cdots \quad A_n \downarrow \mathsf{true}}{\{\{\{A_p\}\,l : T_p, \ldots\}@(p,n)\}_{p\in\{1..n\}} \to \{T_p@(p,n)\}_{p\in\{1..n\}}}$$

Then we have subject reduction.

Theorem 5.3.2 (Subject Reduction)**.**
If $\mathsf{true};\Gamma \vdash P \rhd_{\tilde{s}} \Delta$, Δ *coherent and* $P \to P'$ *then* $\mathsf{true};\Gamma \vdash P' \rhd_{\tilde{s}} \Delta'$ *where* $\Delta \to^{0/1} \Delta'$.

PROOF: By induction on the derivation of $P \to P'$.

5.4 IMPLEMENTATION

APIMS [1] is an implementation of a typechecker and interpretor for the asynchronour π-calculus with multiparty sessions and symmetric synchronisation [96]. We have extended APIMS with support for propositional assertions. This means that only boolean variables and constants can be used in the assertions, and this ensures that assertions validity is decidable. In order to automatically verify the assertions are respected, we have implemented an automated theorem prover for the classical propositional logic. We have implemented proof-search in two axiomatizations: LK [49] and CFLKF [57]. Preliminary testing shows that although the CFLKF has the best runtime on large assertions, the LK based theorem prover has the best runtime for average programs, because a very large part of the theorems proved are trivial, and for these theorems the LK based theorem prover is more efficient.

5.5 RELATED AND FUTURE WORK

The theorem provers we have implemented are based on the LK and CFLKF proof systems, but the is a vast abundance of theorem provers available [103] [71] which can enable both more efficient verification (in practice) and more expressive assertion languages. We can even use a resolution [112] based theorem prover or indeed any method that can decide assertion validity, as we do not currently use the derivations for anything. The results we have proved are not as powerful as the ones proved for the original assertions paper [20]. This is because we do not include the assertions in the programs, and therefore assertion violations are not detected during execution which means we cannot prove the *well typed terms do not go wrong* result. This is not related to the extension with symmetric sum types, and thus it should be possible to extend the program syntax with assertions and prove the result.

5.6 CONCLUSIONS

We have successfully merged the symmetric sum types and the type assertions extensions of the multiparty session types into a single type language. This enables the benefits of assertion types – such as value restrictions and more efficient representation of some interaction patterns – in the workflows that can be represented using symmetric sum types. The extended typing judgement still ensure subject reduction. We have implemented the used language and type verification for the assertion language of classical propositional logic.

Acknowledgements

The first author is supported by the *TrustCare* project, funded by the Danish Strategic Research Agency, Grant #2106-07-0019. The last two authors are partially supported by EPSRC EP/F003757, EP/F002114, EP/G015635 and EP/G015481.

Part IV

APPENDIX

A INTRODUCTION

A.1 PROCESS MATRIX

A.1.1 *Oncology example*

ID	Name	CP	D	N1	N2	PA	Seq	Log	Condition	Input
1.1	**PatientInfo**									
1.1.1	BasicInfo	R	W	W	R	N				patient
1.1.2	LaboratoryResults	R	W	W	R	N				labresults
1.1.3	PatientHistory	R	W	W	R	N				history
1.2	**OrderAndPrepare**						1.1			
1.2.01	CalculateDosis	R	W	R	R	N				data
1.2.02	Sign1	R	W	R	R	N		1.2.01		trustO
1.2.03	VerifyOrder	R	W	R	R	N	1.2.02		(not trustO)	trustO
1.2.04	ControlCalculation	W	R	R	R	N		1.2.02 1.2.03		trustO
1.2.05	MakePreparation	R	N	N	N	W		1.2.04		data
1.2.06	Sign2	R	N	N	N	W		1.2.05		data
1.2.07	VerifyPreparation	R	N	N	N	W	1.2.06		(not trustPA)	data
1.2.08	CheckOutPreparation	W	R	R	R	R		1.2.06 1.2.07		trustPA
1.2.09	Sign3	W	R	R	R	R		1.2.08		data
1.2.10	VerifyCheckout	W	R	R	R	R	1.2.09		(not trustP)	data
1.2.11	CheckOrderMatchesPatient1	N	R	W	R	N		1.2.09 1.2.10		trustO, trustP
1.2.12	CheckOrderMatchesPatient2	N	R	R	W	N		1.2.09 1.2.10		data
1.2.13	Approve	R	W	W	R	R		1.2.11 1.2.12		trustO, trustP
1.3	**MedicineAdministration**						1.2			
1.3.1	AdministerPreparation	R	R	W	W	N				data

A.2 SESSION TYPES

A.2.1 *guisync rules*

(†) = The user for each participant p has accepted the chosen branch (h) and given the used input $\tilde{v}_{hp}$

$$\frac{[\textsc{GuiSync}] \qquad h \in \bigcap_{p=1}^{n} L_p \qquad \vdash \tilde{v}_{hp} : \tilde{S}_{hp}}{\begin{array}{lcl} \texttt{guisync}_{\tilde{s},n,1}\{l(\tilde{x}_{l1} : \tilde{S}_{l1}) : P_{l1}\}_{l \in L_1} & & P_{h1}[\tilde{v}_{h1}/\tilde{x}_{h1}] \\ \mid \ \ldots & \rightarrow & \mid \ \ldots \\ \mid \ \texttt{guisync}_{\tilde{s},n,n}\{l(\tilde{x}_{ln} : \tilde{S}_{ln}) : P_{ln}\}_{l \in L_n} & & \mid \ P_{hn}[\tilde{v}_{hn}/\tilde{x}_{hn}] \end{array}} \ (\dagger)$$

$$\frac{[\textsc{GuiSync}] \quad \forall l \in L'' : \Gamma, \tilde{x}_l : \tilde{S}_l \vdash P_l \triangleright \Delta, \tilde{s} : T_l @ (p, n) \quad L'' \subseteq L \cup L' \quad L' \subseteq L''}{\Gamma \vdash \texttt{guisync}_{\tilde{s},n,p}\{l(\tilde{x}_l : \tilde{S}_l) : P_l\}_{l \in L''} \triangleright \Delta, \tilde{s} : \{l : T_l\}_{l \in L;L'} @ (p, n)}$$

A.2.2 *Oncology example*

```
// DRUG: CHANNEL, TYPE AND IMPLEMENTATION {{{
// GLOBAL TYPE: $drug // {{{
// This is the Interface of a Drug
// Users:
//  1: The drug itself
//  2: User/Owner
define $drug_atadminister =  // Behavior when sent from Authorised person to Nurse
  2=>1:1
  {^Administer:              // The drug is administered to the patient
    Gend
  }
in define $drug_atcheck2 =   // Behavior when sent crom CP to Nurse via porter
  rec $atcheck2.
  2=>1:1
  {^Check:                   // Check Patient Information
    1=>2:2<String>;          // Read Content description
    1=>2:2<String>;          // Read Patient Information
    2=>1:1                   // Verify or Reject
    {^Reject:                // the content does not match prescription
      $atcheck2,             // The drug is destroyed, and the preparation must be
          restarted
     ^Verify:                // The content matches the prescription
      2=>1:1
      {^Check:               // Check Patient Information
        1=>2:2<String>;      // Read Content description
        1=>2:2<String>;      // Read Patient Information
        2=>1:1               // Verify or Reject
```

```
          {^Reject:            // the content does not match prescription
            $atcheck2,         // The drug is destroyed, and the preparation must be
                 restarted
           ^Verify:            // The content matches the prescription
            $drug_atadminister
          }
        }
      },
     ^Destroy:
      Gend
    }
  in define $drug_ataccept1 =  // Behavior when sent from PA to CP
    2=>1:1                     // Verify or Reject
    {^Destroy:                 // the content does not match prescription
      Gend,                    // The drug is destroyed, and the preparation must be
           restarted
     ^Verify:                  // The content matches the prescription
      $drug_atcheck2
    }
  in define $drug_atcheck1 =   // Behavior when sent from PA to CP
    2=>1:1
    {^Check:                   // Check Patient Information
      1=>2:2<String>;          // Read content description
      1=>2:2<String>;          // Read Patient Information
      $drug_ataccept1
    }
  in define $drug =            // Full behavior of a drug
    2=>1:1
    {^Prepare:
      2=>1:1<String>;          // Attach Content Information
      2=>1:1<String>;          // Attach Patient Information
      $drug_atcheck1
    }
  // Used positions: // {{{
  in define $drug_ataccept3 =  // Behavior when sent from PA to CP
    2=>1:1                     // Verify or Reject
    {^Reject:                  // the content does not match prescription
      $drug_atcheck2,          // The drug is destroyed, and the preparation must be
           restarted
     ^Verify:                  // The content matches the prescription
      $drug_atadminister
    }
  in define $drug_atcheck3 =   // Behavior when sent from Nurse to Authorised person
    2=>1:1
    {^Check:                   // Check Patient Information
      1=>2:2<String>;          // Read content description
      1=>2:2<String>;          // Read Patient Information
      $drug_ataccept3
    }
  in define $drug_ataccept2 =
    2=>1:1                     // Verify or Reject
```

```
    {^Reject:                  // the content does not match prescription
      $drug_atcheck2,          // The drug is destroyed, and the preparation must be
          restarted
     ^Verify:                  // The content matches the prescription
      $drug_check3
    } // }}}
  in // }}}
  (nu drug: $drug)
82 // This is the Drug Service Implementation
  def Drug() = // {{{
    link(2,drug,s,1);
    guivalue(2,s,1,"User","Drug");
    ( Drug()
    | s[1]>>
      {^Prepare:
        s[1]>>content;
        s[1]>>patient;
        s[1]>>
92      {^Check:
          s[2]<<content;
          s[2]<<patient;
          s[1]>>
          {^Destroy: end,
           ^Verify:
            def Drug_Check2(s: $drug_atcheck2@(1of2)) =
              s[1]>>
              {^Check:
                s[2]<<content;
102             s[2]<<patient;
                s[1]>>
                {^Reject: Drug_Check2(s),
                 ^Verify:
                  s[1]>>
                  {^Check:
                    s[2]<<content;
                    s[2]<<patient;
                    s[1]>>
                    {^Reject: Drug_Check2(s),
112                  ^Verify:
                      s[1]>>
                      {^Administer: end
                      }
                    }
                  }
                },
               ^Destroy: end
              }
            in Drug_Check2(s)
122       }
        }
      }
```

```
  )
in (Drug() | // }}}
// }}}
// FLOWCHART: CHANNEL, TYPE AND IMPLEMENTATION {{{
// GLOBAL TYPE: $flowchart {{{
// This is the Interface of a Flowchart
// Users:
//  1: The Flowchart Object
//  2: User/Owner
define $flowchart_atsign =
  2=>1:1
  {^Sign:                          // The resonsible nurse signs
    Gend
  }
in define $flowchart_atcheck3 =
  rec $atcheck3.
  2=>1:1
  {^Check:                         // BEGIN: Responsible Nurse Check
    1=>2:2<String>;                // Read Patient Info
    1=>2:2<String>;                // Read Dosis
    2=>1:1
    {^Reject:                      // The prescription and prepared dosis do not
        match
      $atcheck3,
     ^Accept:                      // The prescription and prepared dosis match
      2=>1:1
      {^Check:                     // BEGIN: Authorized Person Check
        1=>2:2<String>;            // Read Patient Info
        1=>2:2<String>;            // Read Dosis
        2=>1:1
        {^Reject:                  // The prescription and prepared dosis do not
            match
          $atcheck3,               // Put drug back, reverification is necessary
         ^Accept:                  // The prescription and prepared dosis match
          $flowchart_atsign
        }
      }
    }
  }
in define $flowchart_atcheck4 =
  2=>1:1
  {^Check:                         // BEGIN: Authorized Person Check
    1=>2:2<String>;                // Read Patient Info
    1=>2:2<String>;                // Read Dosis
    2=>1:1
    {^Reject:                      // The prescription and prepared dosis do not
        match
      $flowchart_atcheck3,         // Put drug back, reverification is necessary
     ^Accept:                      // The prescription and prepared dosis match
      $flowchart_atsign
    }
```

```
  }
// GLOBAL TYPE: $oncology_administer // {{{
// The administering of a drug
// Users:
//  1: The Patient
//  2: The Nurse
//  3: The Authorised Person
in define $oncology_administer =
  1=>2:2<String>;  // Patient gives info to nurse
  {^Reject:
    Gend,
   #Verify:
    1=>3:3<String>;                    // Patient gives details
    2=>3:4<$flowchart_atcheck4@(2of2)>; // Transfer flowchart to verify
    2=>3:4<$drug_atcheck3@(2of2)>;      // Transfer drug to verify
    {^Reject:
      3=>2:2<$flowchart_atcheck3@(2of2)>;
      3=>2:2<$drug_atcheck2@(2of2)>;// Transfer drug to verify
      Gend,
     #Verify:
      3=>2:2<$flowchart_atsign@(2of2)>;
      3=>2:2<$drug_atadminister@(2of2)>;// Transfer drug to verify
      {^Administer:
        Gend
      }
    }
  }// }}}
in define $flowchart_attonurse =
  1=>2:2< <$oncology_administer> >;  // The Nurse reads the contact info
  $flowchart_atcheck3
in define $flowchart_atcheck2 =
  rec $atcheck2.
  2=>1:1
  {^CheckDrug:                       // BEGIN: CP check PA work
    1=>2:2<String>;                  // Read Dosis
    1=>2:2<String>;                  // Read Patient
    2=>1:1
    {^RejectDrug:                    // The Patient Info did not match
      $atcheck2,                     // Try again when new drug is ready
     ^AcceptDrug:                    // The produced drug is accepted
      2=>1:1
      {^Sign:                        // The Controlling Pharmacist signs
        $flowchart_attonurse
      }
    }
  }
in define $flowchart_ataccept2 =
  2=>1:1
  {^RejectDrug:                      // The Patient Info did not match
    $flowchart_atcheck2,             // Try again when new drug is ready
   ^AcceptDrug:                      // The produced drug is accepted
```

```
      2=>1:1
      {^Sign:                          // The Controlling Pharmacist signs
        $flowchart_attonurse
      }
    }
  in define $flowchart_atcalc =
    rec $atcalc.
    2=>1:1
232 {^CalcDosis:
      2=>1:1<String>;                          // Write Dosis
      2=>1:1
      {^Sign:                                  // The Doctor signs
        2=>1:1
        {^Check:                               // BEGIN: CP checks Prescription and
            drug
          1=>2:2<String>;                      // Read Patient Info
          1=>2:2<Int>;                         // Read Patient Height
          1=>2:2<Int>;                         // Read Patient Weight
          1=>2:2<String>;                      // Read Patient History
242       1=>2:2<String>;                      // Read Latest Lab Results
          1=>2:2<String>;                      // Read Dosis
          rec $ataccept1.
          2=>1:1
          {^RejectCalc:
            $atcalc,
           ^AcceptCalc:
            rec $ataccept2.
            2=>1:1
            {^RejectDrug:                      // Calculations are wrong
252           $ataccept2,                      // The calculations must be redone
             ^AcceptDrug:
              2=>1:1
              {^Sign:                          // The Controlling Pharmacist signs
                $flowchart_attonurse
              }
            },
           ^RejectDrug:
            $ataccept1,
           ^AcceptDrug:
262         2=>1:1
            {^RejectCalc:                      // Calculations are wrong
              $atcalc,                         // The calculations must be redone
             ^AcceptCalc:
              2=>1:1
              {^Sign:                          // The Controlling Pharmacist signs
                $flowchart_attonurse
              }
            }
          }
272     }
      }
```

```
  }
in define $flowchart_ataccept1no2 =
  2=>1:1
  {^RejectCalc:                                  // Calculations are wrong
    $flowchart_atcalc,                           // The calculations must be redone
   ^AcceptCalc:
    2=>1:1
    {^Sign:                                      // The Controlling Pharmacist signs
      $flowchart_attonurse
    }
  }
in define $flowchart_ataccept2no1 =
  rec $ataccept2.
  2=>1:1
  {^RejectDrug:                                  // Calculations are wrong
    $ataccept2,                                  // The calculations must be redone
   ^AcceptDrug:
    2=>1:1
    {^Sign:                                      // The Controlling Pharmacist signs
      $flowchart_attonurse
    }
  }
in define $flowchart_ataccept1 =
  rec $ataccept1.
  2=>1:1
  {^RejectCalc:
    $flowchart_atcalc,
   ^AcceptCalc:
    $flowchart_ataccept2no1,
   ^RejectDrug:
    $ataccept1,
   ^AcceptDrug:
    $flowchart_ataccept1no2
  }
in define $flowchart_atcheck1 =
  2=>1:1
  {^Check:
    1=>2:2<String>;                              // Read Patient Info
    1=>2:2<Int>;                                 // Read Patient Height
    1=>2:2<Int>;                                 // Read Patient Weight
    1=>2:2<String>;                              // Read Patient History
    1=>2:2<String>;                              // Read Latest Lab Results
    1=>2:2<String>;                              // Read Dosis
    $flowchart_ataccept1
  }
in define $flowchart =
  2=>1:1
  {^RegisterPatient:
    2=>1:1<String>;                              // Write Patient Info
    2=>1:1<Int>;                                 // Write Patient Height
    2=>1:1<Int>;                                 // Write Patient Weight
```

```
      2=>1:1<String>;                          // Write Patient History
      2=>1:1<String>;                          // Write Latest Lab results
      2=>1:1< <$oncology_administer> >;        // Write contact info
      $flowchart_atcalc
    }
  in // }}}
  (nu flowchart: $flowchart)
// This is the Flowchart Service Implementation
  def Flowchart() = // {{{
    link(2,flowchart,s,1);
    guivalue(2,s,1,"User","Flowchart");
    ( Flowchart()
    | s[1]>>
      {^RegisterPatient:
        s[1]>>patient;
        s[1]>>height;
        s[1]>>weight;
        s[1]>>history;
        s[1]>>labresult;
        s[1]>>channel;
        def AtCalc(s:$flowchart_atcalc@(1of2)) =
        def AtAdminister(patient:String, // {{{
                         dosis:String,
                         s:$flowchart_atcheck3@(1of2)) =
          s[1]>>
          {^Check:
            s[2]<<patient;
            s[2]<<dosis;
            s[1]>>
            {^Reject:
              AtAdminister(patient,dosis,s),
             ^Accept:
              s[1]>>
              {^Check:
                s[2]<<patient;
                s[2]<<dosis;
                s[1]>>
                {^Reject:
                  AtAdminister(patient,dosis,s),
                 ^Accept:
                  s[1]>>
                  {^Sign:
                    end
                  }
                }
              }
            }
          }
        in // }}}
        def AtAccept1no2(patient:String, // {{{
                         dosis:String,
```

```
                    s:$flowchart_ataccept1no2@(1of2)) =
  s[1]>>
  {^RejectCalc:
    AtCalc(s),
   ^AcceptCalc:
    s[1]>>
    {^Sign:
      s[2]<<channel;
      AtAdminister(patient,dosis,s)
    }
  }
in // }}}
def AtAccept2no1(patient:String, // {{{
                 dosis:String,
                 s:$flowchart_ataccept2no1@(1of2)) =
  s[1]>>
  {^RejectDrug:
    AtAccept2no1(patient,dosis,s),
   ^AcceptDrug:
    s[1]>>
    {^Sign:
      s[2]<<channel;
      AtAdminister(patient,dosis,s)
    }
  }
in // }}}
def AtAccept1(patient:String, // {{{
              dosis:String,
              s:$flowchart_ataccept1@(1of2)) =
  s[1]>>
  {^RejectCalc:
    AtCalc(s),
   ^AcceptCalc:
    AtAccept2no1(patient,dosis,s),
   ^RejectDrug:
    AtAccept1(patient,dosis,s),
   ^AcceptDrug:
    AtAccept1no2(patient,dosis,s)
  }
in // }}}
  s[1]>>
  {^CalcDosis:
    s[1]>>dosis;
    s[1]>>
    {^Sign:
      s[1]>>
      {^Check:
        s[2]<<patient;
        s[2]<<height;
        s[2]<<weight;
        s[2]<<history;
```

```
                s[2]<<labresult;
                s[2]<<dosis;
                AtAccept1(patient,dosis,s)
              }
            }
          }
        in AtCalc(s)
      }
    )
in (Flowchart() | // }}}
// }}}
// PORTERS AND WARD INTERFACES {{{
// GLOBAL TYPE: $pharmacy_reception // {{{
// The reception of a flowchart at the pharmacy
// Users:
//  1: The Porter
//  2: The Controlling Pharmacist
define $pharmacy_reception =
  1=>2:2<$flowchart_atcheck1@(2of2)>;                // Porter delivers flowchart to
       CP
  Gend
in // }}}
(nu drug_order: $pharmacy_reception)
(nu pharmacy_reception: $pharmacy_reception)
def Porter_Order() = // {{{
  link(2,drug_order,s,2);
  guivalue(2,s,2,"User","Porter1");
  ( Porter_Order()
  | s[2]>>fc; // Receive flowchart
    link(2,pharmacy_reception,t,1);
    guivalue(2,t,1,"User","Porter1");
    t[2]<<fc; // Deliver flowchart
    end
  )
in ( Porter_Order() | // }}}
// GLOBAL TYPE: $transfer_rejected // {{{
// The sending of a rejected flowchart from the pharmacy
// Users:
//  1: The Controlling Pharmacist
//  2: The Porter
define $transfer_rejected =
  1=>2:2<$flowchart_atcalc@(2of2)>;                // Porter receives flowchart
  Gend
in // }}}
(nu rejected_send: $transfer_rejected)
(nu rejected_receive: $transfer_rejected)
def Porter_Rejected() = // {{{
  link(2,rejected_send,s,2);
  guivalue(2,s,2,"User","Porter2");
  ( Porter_Rejected()
  | s[2]>>fc; // Receive rejected flowchart
```

```
    link(2,rejected_receive,t,1);
  guivalue(2,t,1,"User","Porter2");
    t[2]<<fc; // Deliver rejected flowchart
    end
  )
in ( Porter_Rejected() | // }}}
// GLOBAL TYPE: $transfer_accepted // {{{
// The sending of an accepted flowchart (and the drug) from the pharmacy
// Users:
//  1: The Controlling Pharmacist
//  2: The Porter
define $transfer_accepted =
  1=>2:2<$flowchart_attonurse@(2of2)>;                  // Porter receives flowchart
  1=>2:2<$drug_atcheck2@(2of2)>;                        // Porter receives drug
  Gend
in // }}}
(nu accepted_send: $transfer_accepted)
(nu accepted_receive: $transfer_accepted)
def Porter_Accepted() = // {{{
  link(2,accepted_send,s,2);
  guivalue(2,s,2,"User","Porter3");
  ( Porter_Accepted()
  | s[2]>>theFlowchart;   // Receive flowchart
    s[2]>>theDrug;        // Receive drug
    link(2,accepted_receive,t,1);
    guivalue(2,t,1,"User","Porter3");
    t[2]<<theFlowchart;   // Deliver flowchart
    t[2]<<theDrug;        // Deliver drug
    end
  )
in ( Porter_Accepted() | // }}}
// }}}
// ONCOLOGY WARD {{{
// GLOBAL TYPE: $oncology_reception {{{
// The reception of a patient
// Users:
//  1: The Patient
//  2: The Doctor
define $oncology_reception =
  1=>2:2<String>;
  {^Register:               // Receive Patient (register information)
    2=>1:1< <$oncology_administer> >;
    {^CalcDosis:
      Gend
    }
  }
in // }}}
(nu oncology_reception: $oncology_reception)
def OncologyDoctor() = // {{{
  link(2,oncology_reception,pa,2);
  guivalue(2,pa,2,"User","Oncology Doctor");
```

```
  ( OncologyDoctor()
  | pa[2]>>patient_info;
    guisync(2,pa,2)
    {^Register(patient:String=patient_info,
               height:Int=0,
               weight:Int=0,
               history:String="",
               labresult:String=""):
      link(2,flowchart,fc,2);
      guivalue(2,fc,2,"User","Oncology Doctor");
      fc[1]<<^RegisterPatient;
      fc[1]<<patient;
      fc[1]<<height;
      fc[1]<<weight;
      fc[1]<<history;
      fc[1]<<labresult;
      (nu ref: $oncology_administer)
      fc[1]<<ref;
      pa[1]<<ref;
      guisync(2,pa,2)
      {^CalcDosis(dosis:String=""):
        fc[1]<<^CalcDosis;
        fc[1]<<dosis;
        fc[1]<<^Sign;
        link(2,drug_order,porter,1);
        guivalue(2,porter,1,"User","Oncology Doctor");
        porter[2]<<fc;
        end
      }
    }
  )
in ( OncologyDoctor() | // }}}
(nu oncology_findauth: 1=>2:2< <$oncology_administer> >;Gend)
def OncologyNurse() = // {{{
  link(2,accepted_receive,porter,2);
  guivalue(2,porter,2,"User","Oncology Nurse");
  ( OncologyNurse()
  | def ON_Checks(fc: $flowchart_atcheck3@(2of2), // {{{
                  dr: $drug_atcheck2@(2of2),
                  ref: <$oncology_administer>) =
      link(3,ref,pa,2);     // Find patient from reference
      guivalue(3,pa,2,"User","Oncology Nurse");
      pa[2]>>patient_info;
      guivalue(3,pa,2,"Patient/Info",patient_info);
      fc[1]<<^Check;
      fc[2]>>fc_dosis;
      guivalue(3,pa,2,"Flowchart/Dosis",fc_dosis);
      fc[2]>>fc_patient;
      guivalue(3,pa,2,"Flowchart/Patient",fc_patient);
      dr[1]<<^Check;
      dr[2]>>drug_dosis;
```

```
        guivalue(3,pa,2,"Drug/Dosis",drug_dosis);
        dr[2]>>drug_patient;
        guivalue(3,pa,2,"Drug/Patient",drug_patient);
582     guisync(3,pa,2)
        {^Reject(reason:String=""):
          fc[1]<<^Reject;
          dr[1]<<^Reject;
          ON_Checks(fc,dr,ref),
         #Verify(comment:String=""):
          fc[1]<<^Accept;
          pa[4]<<fc;
          dr[1]<<^Verify;
          pa[4]<<dr;
592       sync(3,pa)
          {^Reject: // Authorized person rejects
            pa[2]>>fc;
            pa[2]>>dr;
            ON_Checks(fc,dr,ref),
           #Verify: // Authorized person accepts
            pa[2]>>fc;
            pa[2]>>dr;
            guisync(3,pa,2)
            {^Administer(comment:String=""):
602           fc[1]<<^Sign;
              dr[1]<<^Administer;
              end
            }
          }
        }
      in // }}}
      porter[2]>>fc;          // Receive Flowchart
      porter[2]>>dr;          // Receive Drug
      fc[2]>>ref;             // Read patient reference from flowchart
612   link(2,oncology_findauth,auth,1);
      guivalue(2,auth,1,"User","Oncology Nurse");
      auth[2]<<ref;           // Send patient reference to auth person
      ON_Checks(fc,dr,ref)
    )
  in ( OncologyNurse() | // }}}
  def OncologyAuthorised() = // {{{
    link(2,oncology_findauth,s,2);
    guivalue(2,s,2,"User","Oncology Auth");
    ( OncologyAuthorised()
622 | def OA_Checks(ref: <$oncology_administer>) =
        link(3,ref,pa,3);     // Find patient from reference
        guivalue(3,pa,3,"User","Oncology Auth");
        sync(3,pa)
        {^Reject:
          OA_Checks(ref),
         #Verify:
          pa[3]>>patient_info;
```

```
            pa[4]>>fc;
            pa[4]>>dr;
632         guivalue(3,pa,3,"Patient/Info",patient_info);
            fc[1]<<^Check;
            fc[2]>>flowchart_dosis;
            guivalue(3,pa,3,"Flowchart/Dosis",flowchart_dosis);
            fc[2]>>flowchart_patient;
            guivalue(3,pa,3,"Flowchart/Patient",flowchart_patient);
            dr[1]<<^Check;
            dr[2]>>drug_dosis;
            guivalue(3,pa,3,"Drug/Dosis",drug_dosis);
            dr[2]>>drug_patient;
642         guivalue(3,pa,3,"Drug/Patient",drug_patient);
            guisync(3,pa,3)
            {^Reject(reason:String=""): // Authorized person rejects
              fc[1]<<^Reject;
              dr[1]<<^Reject;
              pa[2]<<fc;
              pa[2]<<dr;
              OA_Checks(ref),
             #Verify(comment:String=""): // Authorized person accepts
              fc[1]<<^Accept;
652           dr[1]<<^Verify;
              pa[2]<<fc;
              pa[2]<<dr;
              sync(3,pa)
              {^Administer:
                end
              }
            }
          }
        in
662     s[2]>>ref;
        OA_Checks(ref)
      )
    in ( OncologyAuthorised() | // }}}
     // }}}
    // PHARMACY {{{
    // WORKSLIP CHANNEL, TYPE AND IMPLEMENTATION {{{
    // GLOBAL TYPE: $workslip // {{{
    // This is the Interface of a Working Slip
    // Users:
672 //  1: Workslip object
    //  2: User/Owner
    define $workslip_atverify =
      2=>1:1
      {^ReadInfo:
        1=>2:2<String>;          // Read Patient Information
        1=>2:2<String>;          // Read Product
        1=>2:2<String>;          // Read Batch Number
        1=>2:2<Int>;             // Read Quantity
```

```
      Gend
  }
in define $workslip_atregister =
  2=>1:1
  {^RegisterProducts:
    2=>1:1<String>;          // Set Product
    2=>1:1<String>;          // Set Batch Number
    2=>1:1<Int>;             // Set Quantity
    2=>1:1
    {^Sign:
      $workslip_atverify
    },
   ^Destroy:
    Gend
  }
in define $workslip_atread =
  2=>1:1
  {^ReadDescription:
    1=>2:2<String>;
    $workslip_atregister
  }
in define $workslip =
  2=>1:1
  {^SetDescription:
    2=>1:1<String>;          // Write Description
    $workslip_atread
  }
in // }}}
(nu workslip: $workslip)
// This is the Workslip Service Implementation
def Workslip() = // {{{
  link(2,workslip,s,1);
  guivalue(2,s,1,"User","Workslip");
  ( Workslip()
  | s[1]>>
    {^SetDescription:
      s[1]>>description;
      s[1]>>
      {^ReadDescription:
        s[2]<<description;
        s[1]>>
        {^RegisterProducts:
          s[1]>>product;
          s[1]>>batch;
          s[1]>>quantity;
          s[1]>>
          {^Sign:
            s[1]>>
            {^ReadInfo:
              s[2]<<description;
              s[2]<<product;
```

```
732               s[2]<<batch;
                  s[2]<<quantity;
                  end
                }
              },
             ^Destroy:
              end
            }
          }
        }
742   )
    in (Workslip() | // }}}
    // }}}
    // GLOBAL TYPE: $pharmacy // {{{
    // The cooperation at the pharmacy
    // Users:
    //  1: The Controlling Pharmacist
    //  2: The Pharmacy Assistant
    define $pharmacy2 =
      rec $x1.
752   {^MakeWorkslip:
        1=>2:2<$workslip_atread@(2of2)>;          // CP sends workslip to PA
        {^MakeDrug:                               // PA produces drug and sends it to CP
          2=>1:1<$workslip_atverify@(2of2)>;      // PA sends workslip to CP
          2=>1:1<$drug_atcheck1@(2of2)>;          // PA sends drug to CP
          {#AcceptDrug:
            Gend,                                 // The cooperation is complete
           ^RejectDrug:
            $x1                                   // Make new drug
          }
762     }
      }
    in define $pharmacy_mkdr2 =                   // At MakeDrug (Accepted Dosis)
      {^MakeDrug:                                 // PA produces drug and sends it to CP
        2=>1:1<$workslip_atverify@(2of2)>;        // PA sends workslip to CP
        2=>1:1<$drug_atcheck1@(2of2)>;            // PA sends drug to CP
        {#AcceptDrug:
          Gend,                                   // The cooperation is complete
         ^RejectDrug:
          $pharmacy2                              // Make new drug
772     }
      }
    in define $pharmacy_acdr2 =                   // At AcceptDrug (Accepted Dosis)
      {#AcceptDrug:
        Gend,                                     // The cooperation is complete
       ^RejectDrug:
        $pharmacy2                                // Make new drug
      }
    in define $pharmacy =
      rec $x0.
782   {^RejectDosis:
```

```
      Gend,                                        // No internal communication
     #AcceptDosis:
      $pharmacy2,
     #MakeWorkslip:
      1=>2:2<$workslip_atread@(2of2)>;             // CP sends workslip to PA
      {^RejectDosis:
        Gend,
       #AcceptDosis:
        $pharmacy_mkdr2,
792    #MakeDrug:
        2=>1:1<$workslip_atverify@(2of2)>;         // PA sends workslip to CP
        2=>1:1<$drug_atcheck1@(2of2)>;             // PA sends drug to CP
        {^RejectDosis:
          Gend,
         #AcceptDosis:
          $pharmacy_acdr2,
         #AcceptDrug:
          {^RejectDosis:
            Gend,
802          #AcceptDosis:
            Gend
          },
         #RejectDrug:
          $x0
        }
      }
    }
  in define $pharmacy_mkdr =
    {^RejectDosis:
812   Gend,
     #AcceptDosis:
      $pharmacy_mkdr2,
     #MakeDrug:
      2=>1:1<$workslip_atverify@(2of2)>;               // PA sends workslip to CP
      2=>1:1<$drug_atcheck1@(2of2)>;                   // PA sends drug to CP
      {^RejectDosis:
        Gend,
       #AcceptDosis:
        $pharmacy_acdr2,
822    #AcceptDrug:
        {^RejectDosis:
          Gend,
         #AcceptDosis:
          Gend
        },
       #RejectDrug:
        $pharmacy
      }
    }
832 in define $pharmacy_acdr =
    {^RejectDosis:
```

```
      Gend,
     #AcceptDosis:
      $pharmacy_acdr2,
     #AcceptDrug:
      {^RejectDosis:
        Gend,
       #AcceptDosis:
        Gend
      },
     #RejectDrug:
      $pharmacy
    }
  in define $pharmacy_acca =
    {^RejectDosis:
      Gend,
     #AcceptDosis:
      Gend
    }
  // *** VERSION USING STATE *** define $pharmacy = // {{{
  // *** VERSION USING STATE ***   rec $state<ad:Bool=false,mw:Bool=false,mdr:Bool=false
      ,adr:Bool=false,rd:Bool=false>.
  // *** VERSION USING STATE ***   {^RejectDosis[[not (ad or rd)]]:
  // *** VERSION USING STATE ***     $state<false,mw,mdr,adr,true>,
  // *** VERSION USING STATE ***    #AcceptDosis[[not (ad or rd)]]:
  // *** VERSION USING STATE ***     $state<true,mw,mdr,adr,false>,
  // *** VERSION USING STATE ***    ^MakeWorkslip[[not (mw or rd)]]:
  // *** VERSION USING STATE ***     1=>2:2<$workslip_atread@(2of2)>;
      // CP sends workslip to PA
  // *** VERSION USING STATE ***     $state<ad,true,mdr,adr,rd>,
  // *** VERSION USING STATE ***    ^MakeDrug[[mw and not (mdr or rd)]]:
  // *** VERSION USING STATE ***     2=>1:1<$workslip_atverify@(2of2)>;
      // PA sends workslip to CP
  // *** VERSION USING STATE ***     2=>1:1<$drug_atcheck1@(2of2)>;
      // PA sends drug to CP
  // *** VERSION USING STATE ***     $state<ad,mw,true,adr,rd>,
  // *** VERSION USING STATE ***    #AcceptDrug[[mdr and not (adr or rd)]]:
  // *** VERSION USING STATE ***     $state<ad,mw,mdr,true,rd>,
  // *** VERSION USING STATE ***    ^RejectDrug[[mdr and not (adr or rd)]]:
  // *** VERSION USING STATE ***     $state<ad,mw,false,false,rd>,
  // *** VERSION USING STATE ***    ^Finish[[(ad and mw and mdr and adr) or rd]]:
  // *** VERSION USING STATE ***     Gend
  // *** VERSION USING STATE ***   } // }}}
  in // }}}
  (nu pharmacy: $pharmacy)
  def PharmacistAssistant() = // {{{
    link(2,pharmacy,s,2);
    guivalue(2,s,2,"User","Pharmacy Assistant");
    ( PharmacistAssistant()
    | def PA_DestroyWorksheet(reason: String, ws: $workslip_atregister@(2of2)) = // {{{
        (nu accept: {^Accept: Gend})
        link(1,accept,s,1);
```

```
      guivalue(1,s,1,"User","Pharmacy Assistant");
      guivalue(1,s,1,"Action:","Destroy Worksheet");
      guivalue(1,s,1,"Reason:",reason);
      guisync(1,s,1)
      {^Accept(comment:String="Worksheet Destroyed"):
        ws[1]<<^Destroy;
        end
      }
    in // }}}
    def PA_Accept(s:$pharmacy2@(2of2)) = // {{{
      guisync(2,s,2)
      {^MakeWorkslip():
        s[2]>>ws;
        ws[1]<<^ReadDescription;
        ws[2]>>description;
        guivalue(2,s,2,"Description:",description);
        guisync(2,s,2)
        {^MakeDrug(patient:String=description,product:String="",batch:String="",
            quantity:Int=0):
          link(2,drug,d,2);
          guivalue(2,d,2,"User","Pharmacy Assistant");
          d[1]<<^Prepare;
          d[1]<<product;
          d[1]<<patient;
          ws[1]<<^RegisterProducts;
          ws[1]<<product;
          ws[1]<<batch;
          ws[1]<<quantity;
          ws[1]<<^Sign;
          s[1]<<ws;
          s[1]<<d;
          sync(2,s)
          {#AcceptDrug:
            end,
           ^RejectDrug:
            PA_Accept(s)
          }
        }
      }
    in // }}}
    def PA_NoAccept(s:$pharmacy@(2of2)) = // {{{
      guisync(2,s,2)
      {^RejectDosis():
        end,
       #AcceptDosis():
        guivalue(2,s,2,"Dosis","Accepted");
        PA_Accept(s),
       #MakeWorkslip():
        s[2]>>ws;
        ws[1]<<^ReadDescription;
        ws[2]>>description;
```

```
          guivalue(2,s,2,"Description:",description);
932       guisync(2,s,2)
          {^RejectDosis():
            PA_DestroyWorksheet("Dosis rejected",ws),
           #AcceptDosis():
            guivalue(2,s,2,"Dosis","Accepted");
            guisync(2,s,2)
            {^MakeDrug(patient:String=description,product:String="",batch:String="",
                quantity:Int=0):
              link(2,drug,d,2);
              guivalue(2,d,2,"User","Pharmacy Assistant");
              d[1]<<^Prepare;
942           d[1]<<product;
              d[1]<<patient;
              ws[1]<<^RegisterProducts;
              ws[1]<<product;
              ws[1]<<batch;
              ws[1]<<quantity;
              ws[1]<<^Sign;
              s[1]<<ws;
              s[1]<<d;
              sync(2,s)
952           {#AcceptDrug:
                end,
               ^RejectDrug:
                PA_Accept(s)
              }
            },
           #MakeDrug(patient:String=description,product:String="",batch:String="",
               quantity:Int=0):
            link(2,drug,d,2);
            guivalue(2,d,2,"User","Pharmacy Assistant");
            d[1]<<^Prepare;
962         d[1]<<product;
            d[1]<<patient;
            ws[1]<<^RegisterProducts;
            ws[1]<<product;
            ws[1]<<batch;
            ws[1]<<quantity;
            ws[1]<<^Sign;
            s[1]<<ws;
            s[1]<<d;
            sync(2,s)
972         {^RejectDosis:
              end,
             #AcceptDosis:
              guivalue(2,s,2,"Dosis","Accepted");
              sync(2,s)
              {#AcceptDrug:
                end,
               ^RejectDrug:
```

```
                PA_Accept(s)
               },
982            #AcceptDrug:
                sync(2,s)
                {^RejectDosis:
                  end,
                 #AcceptDosis:
                  end
                },
               #RejectDrug:
                PA_NoAccept(s)
              }
992         }
          }
        in // }}}
        PA_NoAccept(s)
      )
    in ( PharmacistAssistant() | // }}}
    def ControllingPharmacist() = // {{{
      link(2,pharmacy_reception,s,2);
      guivalue(2,s,2,"User","Controlling Pharmacist");
      ( ControllingPharmacist()
1002  | def CP_AtMakeWorkslip2(ph: $pharmacy2@(1of2), // {{{
                              fc: $flowchart_ataccept2no1@(2of2)) =
          // FIXME: Double Def
        def CP_AtAcceptDrug2(ph: $pharmacy_acdr2@(1of2), // {{{
                             fc: $flowchart_ataccept2no1@(2of2),
                             dr: $drug_ataccept1@(2of2)) =
          guisync(2,ph,1)
          {^RejectDrug(comment:String=""):
            dr[1]<<^Destroy;
            fc[1]<<^RejectDrug;
1012        CP_AtMakeWorkslip2(ph,fc),
           #AcceptDrug(comment:String=""):
            dr[1]<<^Verify;
            fc[1]<<^AcceptDrug;
            fc[1]<<^Sign;
            link(2,accepted_send,porter,1);
            guivalue(2,porter,1,"User","Controlling Pharmacist");
            porter[2]<<fc;
            porter[2]<<dr;
            end
1022      }
        in // }}}
          // FIXME: Double Def
        def CP_AtMakeDrug2(ph: $pharmacy_mkdr2@(1of2), // {{{
                           fc: $flowchart_ataccept2no1@(2of2)) =
          guisync(2,ph,1)
          {^MakeDrug():
            ph[1]>>ws;
            ph[1]>>dr;
```

```
      // Read info from Worksheet
      ws[1]<<^ReadInfo;
      ws[2]>>ws_patient;
      guivalue(2,ph,1,"Workslip/Patient:",ws_patient);
      ws[2]>>ws_product;
      guivalue(2,ph,1,"Workslip/Product:",ws_product);
      ws[2]>>ws_batch;
      guivalue(2,ph,1,"Workslip/Batch:",ws_batch);
      ws[2]>>ws_quantity;
      guivalue(2,ph,1,"Workslip/Quantity:",ws_quantity);
      // Read info from Drug label
      dr[1]<<^Check;
      dr[2]>>dr_content;
      guivalue(2,ph,1,"Drug/Content:",dr_content);
      dr[2]>>dr_patient;
      guivalue(2,ph,1,"Drug/Patient:",dr_patient);
      CP_AtAcceptDrug2(ph,fc,dr)
    }
  in // }}}
    guisync(2,ph,1)
    {^MakeWorkslip(description:String="Patient:, Dosis:"):
      link(2,workslip,ws,2);
      guivalue(2,ws,2,"User","Controlling Pharmacist");
      ws[1]<<^SetDescription;
      ws[1]<<description;
      ph[2]<<ws;
      CP_AtMakeDrug2(ph,fc)
    }
  in // }}}
  def CP_AtAcceptDrug2(ph: $pharmacy_acdr2@(1of2), // {{{
                       fc: $flowchart_ataccept2no1@(2of2),
                       dr: $drug_ataccept1@(2of2)) =
    guisync(2,ph,1)
    {^RejectDrug(comment:String=""):
      dr[1]<<^Destroy;
      fc[1]<<^RejectDrug;
      CP_AtMakeWorkslip2(ph,fc),
     #AcceptDrug(comment:String=""):
      dr[1]<<^Verify;
      fc[1]<<^AcceptDrug;
      fc[1]<<^Sign;
      link(2,accepted_send,porter,1);
      guivalue(2,porter,1,"User","Controlling Pharmacist");
      porter[2]<<fc;
      porter[2]<<dr;
      end
    }
  in // }}}
  def CP_AtMakeDrug2(ph: $pharmacy_mkdr2@(1of2), // {{{
                     fc: $flowchart_ataccept2no1@(2of2)) =
    guisync(2,ph,1)
```

```
1082      {^MakeDrug():
            ph[1]>>ws;
            ph[1]>>dr;
            // Read info from Worksheet
            ws[1]<<^ReadInfo;
            ws[2]>>ws_patient;
            guivalue(2,ph,1,"Workslip/Patient:",ws_patient);
            ws[2]>>ws_product;
            guivalue(2,ph,1,"Workslip/Product:",ws_product);
            ws[2]>>ws_batch;
1092        guivalue(2,ph,1,"Workslip/Batch:",ws_batch);
            ws[2]>>ws_quantity;
            guivalue(2,ph,1,"Workslip/Quantity:",ws_quantity);
            // Read info from Drug label
            dr[1]<<^Check;
            dr[2]>>dr_content;
            guivalue(2,ph,1,"Drug/Content:",dr_content);
            dr[2]>>dr_patient;
            guivalue(2,ph,1,"Drug/Patient:",dr_patient);
            CP_AtAcceptDrug2(ph,fc,dr)
1102      }
        in // }}}
        def CP_AtMakeWorkslip(ph: $pharmacy@(1of2), // {{{
                              fc: $flowchart_ataccept1@(2of2)) =
        def CP_AtAcceptCalc(ph: {^RejectDosis: Gend, #AcceptDosis: Gend}@(1of2), // {{{
                            fc: $flowchart_ataccept1no2@(2of2),
                            dr: $drug_atcheck2@(2of2)) =
          guisync(2,ph,1)
          {^RejectDosis(reason:String=""):
            // FIXME: reason is not communicated
1112        dr[1]<<^Destroy;
            fc[1]<<^RejectCalc;
            link(2,rejected_send,porter,1);
            guivalue(2,porter,1,"User","Controlling Pharmacist");
            porter[2]<<fc;
            end,
           #AcceptDosis(comment:String=""):
            fc[1]<<^AcceptCalc;
            fc[1]<<^Sign;
            link(2,accepted_send,porter,1);
1122        guivalue(2,porter,1,"User","Controlling Pharmacist");
            porter[2]<<fc;
            porter[2]<<dr;
            end
          }
        in // }}}
        def CP_AtAcceptDrug(ph: $pharmacy_acdr@(1of2), // {{{
                            fc: $flowchart_ataccept1@(2of2),
                            dr: $drug_ataccept1@(2of2)) =
          guisync(2,ph,1)
1132      {^RejectDosis(reason:String=""):
```

```
        // FIXME: reason is not communicated
        dr[1]<<^Destroy;
        fc[1]<<^RejectCalc;
        link(2,rejected_send,porter,1);
        guivalue(2,porter,1,"User","Controlling Pharmacist");
        porter[2]<<fc;
        end,
       #AcceptDosis(comment:String=""):
        fc[1]<<^AcceptCalc;
        CP_AtAcceptDrug2(ph,fc,dr),
       #RejectDrug(comment:String=""):
        dr[1]<<^Destroy;
        fc[1]<<^RejectDrug;
        CP_AtMakeWorkslip(ph,fc),
       #AcceptDrug(comment:String=""):
        dr[1]<<^Verify;
        fc[1]<<^AcceptDrug;
        CP_AtAcceptCalc(ph,fc,dr)
      }
    in // }}}
    def CP_AtMakeDrug(ph: $pharmacy_mkdr@(1of2), // {{{
                      fc: $flowchart_ataccept1@(2of2)) =
      guisync(2,ph,1)
      {^RejectDosis(reason:String=""):
        // FIXME: reason is not communicated
        fc[1]<<^RejectCalc;
        link(2,rejected_send,porter,1);
        guivalue(2,porter,1,"User","Controlling Pharmacist");
        porter[2]<<fc;
        end,
       #AcceptDosis(comment:String=""):
        fc[1]<<^AcceptCalc;
        CP_AtMakeDrug2(ph,fc),
       #MakeDrug():
        ph[1]>>ws;
        ph[1]>>dr;
        // Read info from Worksheet
        ws[1]<<^ReadInfo;
        ws[2]>>ws_patient;
        guivalue(2,ph,1,"Workslip/Patient:",ws_patient);
        ws[2]>>ws_product;
        guivalue(2,ph,1,"Workslip/Product:",ws_product);
        ws[2]>>ws_batch;
        guivalue(2,ph,1,"Workslip/Batch:",ws_batch);
        ws[2]>>ws_quantity;
        guivalue(2,ph,1,"Workslip/Quantity:",ws_quantity);
        // Read info from Drug label
        dr[1]<<^Check;
        dr[2]>>dr_content;
        guivalue(2,ph,1,"Drug/Content:",dr_content);
        dr[2]>>dr_patient;
```

```
        guivalue(2,ph,1,"Drug/Patient:",dr_patient);
        CP_AtAcceptDrug(ph,fc,dr)
      }
    in // }}}
      guisync(2,ph,1)
      {^RejectDosis(reason:String=""):
        // FIXME: reason is not communicated
        fc[1]<<^RejectCalc;
        link(2,rejected_send,porter,1);
        guivalue(2,porter,1,"User","Controlling Pharmacist");
        porter[2]<<fc;
        end,
       #AcceptDosis(comment:String=""):
        fc[1]<<^AcceptCalc;
        CP_AtMakeWorkslip2(ph,fc),
       #MakeWorkslip(description:String="Patient: <>, Dosis: <>"):
        link(2,workslip,ws,2);
        guivalue(2,ws,2,"User","Controlling Pharmacist");
        ws[1]<<^SetDescription;
        ws[1]<<description;
        ph[2]<<ws;
        CP_AtMakeDrug(ph,fc)
      }
    in // }}}
    s[2]>>fc;              // Receive Flowchart
    link(2,pharmacy,ph,1); // Create Pharmacy Workflow
    guivalue(2,ph,1,"User","Controlling Pharmacist");
    guivalue(2,ph,1,"User","Controling Pharmacist");
    // Read information from Flowchart
    fc[1]<<^Check;
    fc[2]>>patient;
    guivalue(2,ph,1,"Flowchart/Patient",patient);
    fc[2]>>height;
    guivalue(2,ph,1,"Flowchart/Patient/Height",height);
    fc[2]>>weight;
    guivalue(2,ph,1,"Flowchart/Patient/Weight",weight);
    fc[2]>>history;
    guivalue(2,ph,1,"Flowchart/Patient/History",history);
    fc[2]>>labresult;
    guivalue(2,ph,1,"Flowchart/Patient/Lab. Result",labresult);
    fc[2]>>dosis;
    guivalue(2,ph,1,"Flowchart/Dosis",dosis);
    CP_AtMakeWorkslip(ph,fc)
  )
in (ControllingPharmacist() | // }}}
// }}}
// PATIENT TEMPLATE IMPLEMENTATION {{{
def Patient(myInfo: String) =
  link(2,oncology_reception,doctor,1);
  guivalue(2,doctor,1,"User","Patient");
  doctor[2]<<myInfo;
```

```
    sync(2,doctor)
    {^Register:
      doctor[1]>>ref;
      ( sync(2,doctor)
        {^CalcDosis:
          end
        }
    | def P_Wait(ref: <$oncology_administer>) =
        link(3,ref,s,1);
        guivalue(3,s,1,"User","Patient");
        s[2]<<myInfo;
        sync(3,s)
        {^Reject:
          P_Wait(ref),
         #Verify:
          s[3]<<myInfo;
          sync(3,s)
          {^Reject:
            P_Wait(ref),
           #Verify:
            guisync(3,s,1)
            {^Administer(comment:String=""):
              end
            }
          }
        }
      in P_Wait(ref)
    )
  }
in ( // }}}
// Test Patient:
  Patient("Name: A, CPR: A-xxxx")
) ) ) ) ) ) ) ) ) ) )
```

A.2.3 *Urology example*

```
// Journal {{{
// Jounal Interface {{{
// Participants:
// 1: User
// 2: Journal
define $journal xEnrolled
                xNrOne xNrTwo
                xOperation xInformed xBooked
                xAdmMorning xAdmAfternoon xAdmEvening =
  rec $actions<xEnrolled:Bool=xEnrolled,
               xNrOne:Bool=xNrOne,
               xNrTwo:Bool=xNrTwo,
               xOperation:Bool=xOperation,
               xInformed:Bool=xInformed,
               xBooked:Bool=xBooked,
               xAdmMorning:Bool=xAdmMorning,
               xAdmAfternoon:Bool=xAdmAfternoon,
               xAdmEvening:Bool=xAdmEvening>.
    1=>2:1
    {^ReadName[[xEnrolled]]:
      2=>1:2<String>;
      $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>,
     ^WriteName[[xEnrolled]]:
      1=>2:1<String>;
      $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>,
     ^ReadCPR[[xEnrolled]]:
      2=>1:2<String>;
      $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>,
     ^WriteCPR[[xEnrolled]]:
      1=>2:1<String>;
      $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>,
     ^ReadRoom[[xEnrolled]]:
      2=>1:2<String>;
      $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>,
     ^WriteRoom[[xEnrolled]]:
      1=>2:1<String>;
      $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>,
     ^ReadFluid[[xEnrolled]]:
      2=>1:2<String>;
      $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>,
     ^WriteFluid[[xEnrolled]]:
```

```
 1=>2:1<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^ReadEnergy[[xEnrolled]]:
 2=>1:2<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteEnergy[[xEnrolled]]:
 1=>2:1<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^ReadSymptoms[[xEnrolled]]:
 2=>1:2<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteSymptoms[[xEnrolled]]:
 1=>2:1<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^ReadBooking[[xEnrolled]]:
 2=>1:2<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteBooking[[xEnrolled]]:
 1=>2:1<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^ReadMedication[[xEnrolled]]:
 2=>1:2<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteMedication[[xEnrolled]]:
 1=>2:1<String>;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteNrOne[[xEnrolled]]:
 1=>2:1<Bool> as xNrOne;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteNrTwo[[xEnrolled]]:
 1=>2:1<Bool> as xNrTwo;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteOperation[[xEnrolled]]:
 1=>2:1<Bool> as xOperation;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
^WriteInformed[[xEnrolled]]:
 1=>2:1<Bool> as xInformed;
 $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
     xAdmAfternoon,xAdmEvening>,
```

```
      ^WriteBooked[[xEnrolled]]:
       1=>2:1<Bool> as xBooked;
       $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
           xAdmAfternoon,xAdmEvening>,
      ^WriteAdministration[[xEnrolled]]:
       1=>2:1<Bool> as xAdmMorning;
       1=>2:1<Bool> as xAdmAfternoon;
       1=>2:1<Bool> as xAdmEvening;
       $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
           xAdmAfternoon,xAdmEvening>,
      ^WriteEnrolled:
       1=>2:1<Bool> as xEnrolled;
       $actions<xEnrolled,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
           xAdmAfternoon,xAdmEvening>,
       ^Archive[[not xEnrolled]]: Gend
     }
in // }}}
// Create Channel: journal {{{
(nu journal :
  1=>2:1<String>; // Set Name
  1=>2:1<String>; // Set CPR
  1=>2:1<String>; // Set Room
  1=>2:1<String>; // Set Symptoms
  $journal<true,false,false,false,false,false,false,false,false>) // }}}
( // Journal Service Implementation {{{
  def Journal() =
    link(2,journal,s,2);
    ( Journal()
    | s[1]>>name;
      s[1]>>cpr;
      s[1]>>room;
      s[1]>>symptoms;
      def State<pEnrolled:Bool, pNrOne:Bool, pNrTwo:Bool,
                pOperation:Bool, pInformed:Bool, pBooked:Bool,
                pAdmMorning:Bool,pAdmAfternoon:Bool,
                pAdmEvening:Bool>
               (s:$journal<pEnrolled,pNrOne,pNrTwo,
                           pOperation,pInformed,pBooked,
                           pAdmMorning,pAdmAfternoon,
                           pAdmEvening>@(2of2),
                pName:String, pCPR:String, pRoom:String,
                pFluid:String, pEnergy:String,
                pSymptoms:String, pBooking:String, pMedication:String) =
        s[1]>>
        {^ReadName:
          s[2]<<pName;
          State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
              pAdmAfternoon,pAdmEvening>
               (s,pName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
         ^WriteName:
          s[1]>>newName;
```

```
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,newName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^ReadCPR:
 s[2]<<pCPR;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^WriteCPR:
 s[1]>>newCPR;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,newCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^ReadRoom:
 s[2]<<pCPR;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^WriteRoom:
 s[1]>>newRoom;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,newRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^ReadFluid:
 s[2]<<pFluid;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^WriteFluid:
 s[1]>>newFluid;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,newFluid,pEnergy,pSymptoms,pBooking,pMedication),
^ReadEnergy:
 s[2]<<pEnergy;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^WriteEnergy:
 s[1]>>newEnergy;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,newEnergy,pSymptoms,pBooking,pMedication),
^ReadSymptoms:
 s[2]<<pSymptoms;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
^WriteSymptoms:
 s[1]>>neweSymptoms;
```

```
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,neweSymptoms,pBooking,pMedication),
^ReadBooking:
 s[2]<<pBooking;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pBooking,pBooking,pMedication),
^WriteBooking:
 s[1]>>newBooking;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,newBooking,pBooking,pMedication),
^ReadMedication:
 s[2]<<pMedication;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pMedication,pMedication,pMedication)
          ,
^WriteMedication:
 s[1]>>newMedication;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,newMedication,pMedication,
          pMedication),
^WriteNrOne:
 s[1]>>newNrOne;
 State<pEnrolled,newNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pBooking,pBooking,pMedication),
^WriteNrTwo:
 s[1]>>newNrTwo;
 State<pEnrolled,pNrOne,newNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pBooking,pBooking,pMedication),
^WriteOperation:
 s[1]>>newOperation;
 State<pEnrolled,pNrOne,pNrTwo,newOperation,pInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pBooking,pBooking,pMedication),
^WriteInformed:
 s[1]>>newInformed;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,newInformed,pBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pBooking,pBooking,pMedication),
^WriteBooked:
 s[1]>>newBooked;
 State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,newBooked,pAdmMorning,
     pAdmAfternoon,pAdmEvening>
      (s,pName,pCPR,pRoom,pFluid,pEnergy,pBooking,pBooking,pMedication),
^WriteAdministration:
```

```
            s[1]>>newAdmMorning;
            s[1]>>newAdmAfternoon;
            s[1]>>newAdmEvening;
            State<pEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,newAdmMorning,
                newAdmAfternoon,newAdmEvening>
                 (s,pName,pCPR,pRoom,pFluid,pEnergy,pBooking,pBooking,pMedication),
          ^WriteEnrolled:
            s[1]>>newEnrolled;
            State<newEnrolled,pNrOne,pNrTwo,pOperation,pInformed,pBooked,pAdmMorning,
                pAdmAfternoon,pAdmEvening>
                 (s,pName,pCPR,pRoom,pFluid,pEnergy,pSymptoms,pBooking,pMedication),
          ^Archive: end
         }
       in State<true,false,false,false,false,false,false,false,false>(s,name,cpr,room
            ,"(unknown)","(unknown)",symptoms,"None","None")
    )
  in Journal() | // }}}
// }}}
// Secretary {{{
// Secretary Interface  // {{{
// Participants:
// 1: Nurse
// 2: Secretary
define $fetch_journal xNrOne xNrTwo xOperation xInformed xBooked xAdmMorning
    xAdmAfternoon xAdmEvening =
  2=>1:2<$journal<true,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
      xAdmAfternoon,xAdmEvening>@(1of2)>;
  Gend
in
define $secretary =
  1=>2:1<Bool> as xNrOne;
  1=>2:1<Bool> as xNrTwo;
  1=>2:1<Bool> as xOperation;
  1=>2:1<Bool> as xInformed;
  1=>2:1<Bool> as xBooked;
  1=>2:1<Bool> as xAdmMorning;
  1=>2:1<Bool> as xAdmAfternoon;
  1=>2:1<Bool> as xAdmEvening;
  1=>2:1<$journal<true,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
      xAdmAfternoon,xAdmEvening>@(1of2)>;
  $fetch_journal<xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,xAdmAfternoon,
      xAdmEvening>
in // }}}
// Create Channel: secretary // {{{
(nu secretary: $secretary) // }}}
( // Secretary Service Implementation {{{
  def Secretary() =
    link(2,secretary,s,2);
    s[1]>>xNrOne;
    s[1]>>xNrTwo;
    s[1]>>xOperation;
```

```
      s[1]>>xInformed;
      s[1]>>xBooked;
      s[1]>>xAdmMorning;
      s[1]>>xAdmAfternoon;
      s[1]>>xAdmEvening;
      s[1]>>j;
      // FIXME: Write updates to the journal
      s[2]<<j;
      // Handle next assignment
      Secretary()
    in
      ( Secretary() | Secretary() | Secretary() ) | // Create 3 secretaries }}}
// }}}
// Treatment {{{
// Treatments Interface {{{
// Participants:
// 1: Patient
// 2: Doctor
// 3: Nurse
define $treatment_2114 xNrOne xNrTwo
                       xDischarge
                       xJournalHere xJournalUpdated
                       xOperation xInformed xBooked
                       xAdmMorning xAdmAfternoon xAdmEvening =
  rec $treatment<xNrOne:Bool=xNrOne,
                 xNrTwo:Bool=xNrTwo,
                 xDischarge:Bool=xDischarge,
                 xJournalHere:Bool=xJournalHere,
                 xJournalUpdated:Bool=xJournalUpdated,
                 xOperation:Bool=xOperation,
                 xInformed:Bool=xInformed,
                 xBooked:Bool=xBooked,
                 xAdmMorning:Bool=xAdmMorning,
                 xAdmAfternoon:Bool=xAdmAfternoon,
                 xAdmEvening:Bool=xAdmEvening>.
    {^Rounds[[xJournalHere and not xAdmMorning and not xAdmAfternoon and not
        xAdmEvening]]:
      3=>2:2<$journal<true,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
          xAdmAfternoon,xAdmEvening>@(1of2)>;
      // Symptoms are added to journal
      rec $stuegang<xNrOne:Bool=xNrOne,
                    xNrTwo:Bool=xNrTwo,
                    xDischarge:Bool=xDischarge,
                    xJournalHere:Bool=xJournalHere,
                    xJournalUpdated:Bool=xJournalUpdated,
                    xOperation:Bool=xOperation,
                    xInformed:Bool=xInformed,
                    xBooked:Bool=xBooked,
                    xAdmMorning:Bool=xAdmMorning,
                    xAdmAfternoon:Bool=xAdmAfternoon,
                    xAdmEvening:Bool=xAdmEvening>.
```

```
        {#Surgery:
          2=>1:1<Bool> as newOperation;
          2=>3:3<Bool> as pOperation [[(newOperation or not pOperation) and (not
              newOperation or pOperation)]]; // =newOperation
          2=>1:1<Bool> as newInformed;
          2=>3:3<Bool> as pInformed [[(newInformed or not pInformed) and (not
              newInformed or pInformed)]]; // =newInformed
          $stuegang<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,newOperation
              ,newInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
         #Discharge[[xNrOne and xNrTwo]]:
          2=>1:1<Bool> as newDischarge;
          2=>3:3<Bool> as pDischarge [[(newDischarge or not pDischarge) and (not
              newDischarge or pDischarge)]]; // =newDischarge
          $stuegang<xNrOne,xNrTwo,newDischarge,xJournalHere,xJournalUpdated,xOperation
              ,xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
         #Medication:
          2=>1:1<String>; // Doctor informs about medication
          2=>3:3<String>; // Doctor informs about medication
          2=>1:1<Bool> as newAdmMorning;
          2=>3:3<Bool> as pAdmMorning [[(newAdmMorning or not pAdmMorning) and (not
              newAdmMorning or pAdmMorning)]]; // =newAdmMorning
          2=>1:1<Bool> as newAdmAfternoon;
          2=>3:3<Bool> as pAdmAfternoon [[(newAdmAfternoon or not pAdmAfternoon) and (
              not newAdmAfternoon or pAdmAfternoon)]]; // =newAdmAfternoon
          2=>1:1<Bool> as newAdmEvening;
          2=>3:3<Bool> as pAdmEvening [[(newAdmEvening or not pAdmEvening) and (not
              newAdmEvening or pAdmEvening)]]; // =newAdmEvening
          $stuegang<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
              xInformed,xBooked,newAdmMorning,newAdmAfternoon,newAdmEvening>,
         ^StopRounds:
          2=>3:3<$journal<true,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
              xAdmAfternoon,xAdmEvening>@(1of2)>;
          $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
              xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>
        },
       #Dictate[[xJournalHere]]:
        $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,true,xOperation,xInformed,
            xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
       ^SendJournal[[xJournalHere and xJournalUpdated]]:
        $treatment<xNrOne,xNrTwo,xDischarge,false,xJournalUpdated,xOperation,xInformed,
            xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
       ^FetchJournal[[not xJournalHere]]:
        $treatment<xNrOne,xNrTwo,xDischarge,true,false,xOperation,xInformed,xBooked,
            xAdmMorning,xAdmAfternoon,xAdmEvening>,
       #Move[[xJournalHere]]:
        // New room is written to journal
        $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
       ^VisitMorning[[xJournalHere]]: // Time Constraint?
        //1=>3:4<String>; // Patient explains symptoms
        // Symptoms are added to the journal
```

```
 1=>3:3<Bool> as newNrOne;
 3=>2:2<Bool> as pNrOne [[(newNrOne or not pNrOne) and (not newNrOne or pNrOne)
     ]]; // =newNrOne
 1=>3:3<Bool> as newNrTwo;
 3=>2:2<Bool> as pNrTwo [[(newNrTwo or not pNrTwo) and (not newNrTwo or pNrTwo)
     ]]; // =newNrTwo
 //1=>3:3<String>; // Patient explains how much liquid
 //1=>3:3<String>; // and energy has been consumed
 // Liquid- and energy consumption is registered in journal
 $treatment<newNrOne,newNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
     xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
^VisitAfternoon[[xJournalHere]]: // Time Constraint?
 //1=>3:4<String>; // Patient explains symptoms
 // The symptoms are added to the journal
 1=>3:3<Bool> as newNrOne;
 3=>2:2<Bool> as pNrOne [[(newNrOne or not pNrOne) and (not newNrOne or pNrOne)
     ]]; // =newNrOne
 1=>3:3<Bool> as newNrTwo;
 3=>2:2<Bool> as pNrTwo [[(newNrTwo or not pNrTwo) and (not newNrTwo or pNrTwo)
     ]]; // =newNrTwo
 //1=>3:3<String>; // Patient explains how much liquid
 //1=>3:3<String>; // and energy has been consumed
 // Liquid- and energy consumption is registered in journal
 $treatment<newNrOne,newNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
     xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
^VisitEvening[[xJournalHere]]: // Time Constraint?
 //1=>3:4<String>; // Patient explains symptoms
 // The symptoms are added to the journal
 1=>3:3<Bool> as newNrOne;
 3=>2:2<Bool> as pNrOne [[(newNrOne or not pNrOne) and (not newNrOne or pNrOne)
     ]]; // =newNrOne (not done physically)
 1=>3:3<Bool> as newNrTwo;
 3=>2:2<Bool> as pNrTwo [[(newNrTwo or not pNrTwo) and (not newNrTwo or pNrTwo)
     ]]; // =newNrTwo (not done physically)
 //1=>3:3<String>; // Patient explains how much liquid
 //1=>3:3<String>; // and energy has been consumed
 // Liquid- and energy consumption is registered in journal
 $treatment<newNrOne,newNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
     xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
^AdmMorning[[xJournalHere and xAdmMorning]]: // Time Constraint?
 // Nurse administers the medication to the patient
 $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
     xInformed,xBooked,false,xAdmAfternoon,xAdmEvening>,
^AdmAfternoon[[xJournalHere and xAdmAfternoon]]: // Time Constraint?
 // Nurse administers the medication to the patient
 $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
     xInformed,xBooked,xAdmMorning,false,xAdmEvening>,
^AdmEvening[[xJournalHere and xAdmEvening]]: // Time Constraint?
 // Nurse administers the medication to the patient
 $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
     xInformed,xBooked,xAdmMorning,xAdmAfternoon,false>,
```

```
      ^Book[[xJournalHere and xOperation and not xBooked]]:
       $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
           xInformed,true,xAdmMorning,xAdmAfternoon,xAdmEvening>,
      ^Surgery[[xJournalHere and xOperation and xInformed and xBooked]]:
       {^StopSurgery:
         $treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,false,false,
             false,xAdmMorning,xAdmAfternoon,xAdmEvening>
       },
      ^Discharge[[xDischarge and xJournalHere]]: Gend
     }
in
define $treatment_2114_stuegang xNrOne xNrTwo
                     xDischarge
                     xJournalHere xJournalUpdated
                     xOperation xInformed xBooked
                     xAdmMorning xAdmAfternoon xAdmEvening =
  rec $stuegang<xNrOne:Bool=xNrOne,
                xNrTwo:Bool=xNrTwo,
                xDischarge:Bool=xDischarge,
                xJournalHere:Bool=xJournalHere,
                xJournalUpdated:Bool=xJournalUpdated,
                xOperation:Bool=xOperation,
                xInformed:Bool=xInformed,
                xBooked:Bool=xBooked,
                xAdmMorning:Bool=xAdmMorning,
                xAdmAfternoon:Bool=xAdmAfternoon,
                xAdmEvening:Bool=xAdmEvening>.
    {#Surgery:
      2=>1:1<Bool> as newOperation;
      2=>3:3<Bool> as pOperation [[(newOperation or not pOperation) and (not
          newOperation or pOperation)]]; // =newOperation
      2=>1:1<Bool> as newInformed; // [[newOperation or not newInformed]];
      2=>3:3<Bool> as pInformed [[(newInformed or not pInformed) and (not newInformed
          or pInformed)]]; // =newInformed
      $stuegang<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,newOperation,
          newInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
     #Discharge[[xNrOne and xNrTwo]]:
      2=>1:1<Bool> as newDischarge; // [[(xNrOne and xNrTwo) or not newDischarge]];
      2=>3:3<Bool> as pDischarge [[(newDischarge or not pDischarge) and (not
          newDischarge or pDischarge)]]; // =newDischarge
      $stuegang<xNrOne,xNrTwo,newDischarge,xJournalHere,xJournalUpdated,xOperation,
          xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>,
     #Medication:
      2=>1:1<String>; // Doctor explains the medication
      2=>3:3<String>; // Doctor explains the medication
      2=>1:1<Bool> as newAdmMorning;
      2=>3:3<Bool> as pAdmMorning [[(newAdmMorning or not pAdmMorning) and (not
          newAdmMorning or pAdmMorning)]]; // =newAdmMorning
      2=>1:1<Bool> as newAdmAfternoon;
      2=>3:3<Bool> as pAdmAfternoon [[(newAdmAfternoon or not pAdmAfternoon) and (not
          newAdmAfternoon or pAdmAfternoon)]]; // =newAdmAfternoon
```

```
        2=>1:1<Bool> as newAdmEvening;
        2=>3:3<Bool> as pAdmEvening [[(newAdmEvening or not pAdmEvening) and (not
            newAdmEvening or pAdmEvening)]]; // =newAdmEvening
        $stuegang<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,newAdmMorning,newAdmAfternoon,newAdmEvening>,
       ^StopRounds:
        2=>3:3<$journal<true,xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning,
            xAdmAfternoon,xAdmEvening>@(1of2)>;
        $treatment_2114<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
            ,xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>
      }
in // }}}
// Create Channel: treatment {{{
(nu treatment :
  1=>3:3<$journal<true,false,false,false,false,false,false,false,false>@(1of2)>;
  $treatment_2114<false,false,false,true,false,false,false,false,false,false,false>)
      // }}}
( // Doctor service Implementation {{{
  def Doctor() =
    link(3,treatment,s,2); // Connect as Doctor
    ( Doctor()
    | def Treatment<xNrOne:Bool,xNrTwo:Bool,xDischarge:Bool,
                    xJournalHere:Bool,xJournalUpdated:Bool,
                    xOperation:Bool,xInformed:Bool,xBooked:Bool,
                    xAdmMorning:Bool,xAdmAfternoon:Bool,xAdmEvening:Bool>
              (b:$treatment_2114<xNrOne,xNrTwo,xDischarge,
                       xJournalHere,xJournalUpdated,
                       xOperation,xInformed,xBooked,
                       xAdmMorning,xAdmAfternoon,xAdmEvening>@(2of3))=
        def Rounds<xNrOne:Bool,xNrTwo:Bool,xDischarge:Bool, // {{{
                    xJournalUpdated:Bool,
                    xOperation:Bool,xInformed:Bool,xBooked:Bool,
                    xAdmMorning:Bool,xAdmAfternoon:Bool,xAdmEvening:Bool>
                    (b:$treatment_2114_stuegang<xNrOne,xNrTwo,xDischarge,
                             true,xJournalUpdated,
                             xOperation,xInformed,xBooked,
                             xAdmMorning,xAdmAfternoon,xAdmEvening>@(2of3),
                     j:$journal<true,xNrOne,xNrTwo,
                                xOperation,xInformed,xBooked,
                                xAdmMorning,xAdmAfternoon,xAdmEvening>@(1of2)) =
          guivalue(3,b,2,"NrOne",if xNrOne then "Yes" else "No");
          guivalue(3,b,2,"NrTwo",if xNrTwo then "Yes" else "No");
          guivalue(3,b,2,"Discharged",if xDischarge then "Yes" else "No");
          guivalue(3,b,2,"Journal is here","Yes");
          guivalue(3,b,2,"Journal has changes",if xJournalUpdated then "Yes" else "No
              ");
          guivalue(3,b,2,"Surgeryes",if xOperation then "Yes" else "No");
          guivalue(3,b,2,"Patient is informed",if xInformed then "Yes" else "No");
          guivalue(3,b,2,"OR is booked",if xBooked then "Yes" else "No");
          guivalue(3,b,2,"Administer in the Morning",if xAdmMorning then "Yes" else "
              No");
```

```
  guivalue(3,b,2,"Administer in the Afternoon",if xAdmAfternoon then "Yes"
      else "No");
  guivalue(3,b,2,"Administer in the Evening",if xAdmEvening then "Yes" else "
      No");
  guisync(3,b,2)
  {#Surgery(operer:Bool=xOperation,
            patientinformeret:Bool=false):
    b[1]<<operer;
    b[3]<<operer;
    b[1]<<patientinformeret;
    b[3]<<patientinformeret;
    j[1]<<^WriteOperation;
    j[1]<<operer;
    j[1]<<^WriteInformed;
    j[1]<<patientinformeret;
    Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,operer,patientinformeret,
        xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
   #Discharge[[xNrOne and xNrTwo]](udskriv:Bool=xDischarge):
    b[1]<<udskriv;
    b[3]<<udskriv;
    Rounds<xNrOne,xNrTwo,udskriv,xJournalUpdated,xOperation,xInformed,xBooked,
        xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
   #Medication(medicin:String="",
              morgen:Bool=xAdmMorning,
              eftermiddag:Bool=xAdmAfternoon,
              aften:Bool=xAdmEvening):
    b[1]<<medicin;
    b[3]<<medicin;
    j[1]<<^WriteMedication;
    j[1]<<medicin;
    b[1]<<morgen;
    b[3]<<morgen;
    b[1]<<eftermiddag;
    b[3]<<eftermiddag;
    b[1]<<aften;
    b[3]<<aften;
    j[1]<<^WriteAdministration;
    j[1]<<morgen;
    j[1]<<eftermiddag;
    j[1]<<aften;
    Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
        xBooked,morgen,eftermiddag,aften>(b,j),
   ^StopRounds(kommentar:String=""):
    b[3]<<j;
    Treatment<xNrOne,xNrTwo,xDischarge,true,xJournalUpdated,xOperation,
        xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b)
  } // }}}
in
  guivalue(3,b,2,"NrOne",if xNrOne then "Yes" else "No");
  guivalue(3,b,2,"NrTwo",if xNrTwo then "Yes" else "No");
  guivalue(3,b,2,"Discharged",if xDischarge then "Yes" else "No");
```

```
guivalue(3,b,2,"Journal is here",if xJournalHere then "Yes" else "No");
guivalue(3,b,2,"Journal has changes",if xJournalUpdated then "Yes" else "No
    ");
guivalue(3,b,2,"Surgeryes",if xOperation then "Yes" else "No");
guivalue(3,b,2,"Patient is informed",if xInformed then "Yes" else "No");
guivalue(3,b,2,"OR is booked",if xBooked then "Yes" else "No");
guivalue(3,b,2,"Administer in the Morning",if xAdmMorning then "Yes" else "
    No");
guivalue(3,b,2,"Administer in the Afternoon",if xAdmAfternoon then "Yes"
    else "No");
guivalue(3,b,2,"Administer in the Evening",if xAdmEvening then "Yes" else "
    No");
guisync(3,b,2)
{^Rounds[[xJournalHere and not xAdmMorning and not xAdmAfternoon and not
    xAdmEvening]](kommentar:String=""):
  b[2]>>j;
  Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
      xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
 #Dictate[[xJournalHere]](diktat:String=""):
  // FIXME: Send changes directly to secretary
  Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,true,xOperation,xInformed,
      xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
 ^SendJournal[[xJournalHere and xJournalUpdated]]():
  Treatment<xNrOne,xNrTwo,xDischarge,false,xJournalUpdated,xOperation,
      xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
 ^FetchJournal[[not xJournalHere]]():
  Treatment<xNrOne,xNrTwo,xDischarge,true,false,xOperation,xInformed,xBooked
      ,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
 #Move[[xJournalHere]]():
  Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
      ,xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
 ^VisitMorning[[xJournalHere]]():
  b[2]>>xNrOne;
  b[2]>>xNrTwo;
  Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
      ,xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
 ^VisitAfternoon[[xJournalHere]]():
  b[2]>>xNrOne;
  b[2]>>xNrTwo;
  Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
      ,xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
 ^VisitEvening[[xJournalHere]]():
  b[2]>>xNrOne;
  b[2]>>xNrTwo;
  Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
      ,xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
 ^AdmMorning[[xJournalHere and xAdmMorning]]():
  Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
      ,xInformed,xBooked,false,xAdmAfternoon,xAdmEvening>(b),
 ^AdmAfternoon[[xJournalHere and xAdmAfternoon]]():
```

```
              Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
                  ,xInformed,xBooked,xAdmMorning,false,xAdmEvening>(b),
             ^AdmEvening[[xJournalHere and xAdmEvening]]():
              Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
                  ,xInformed,xBooked,xAdmMorning,xAdmAfternoon,false>(b),
             ^Book[[xJournalHere and xOperation and not xBooked]]():
              Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation
                  ,xInformed,true,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
             ^Surgery[[xJournalHere and xOperation and xInformed and xBooked]](kommentar
                 :String=""):
              guisync(3,b,2)
              {^StopSurgery(kommentar:String=kommentar):
                Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,false,
                    false,false,xAdmMorning,xAdmAfternoon,xAdmEvening>(b)
              },
             ^Discharge[[xDischarge and xJournalHere]](): end
            }
      in
        Treatment<false,false,false,true,false,false,false,false,false,false,false>(s)
    )
  in Doctor() | // }}}
( // Nurse Service Implementation {{{
  def Nurse() =
    link(3,treatment,s,3); // Connect as Nurse
    ( Nurse()
    | def WithJournal<xNrOne:Bool,xNrTwo:Bool,xDischarge:Bool, // {{{
                      xJournalUpdated:Bool,
                      xOperation:Bool,xInformed:Bool,xBooked:Bool,
                      xAdmMorning:Bool,xAdmAfternoon:Bool,xAdmEvening:Bool>
                (b:$treatment_2114<xNrOne,xNrTwo,xDischarge,
                      true,xJournalUpdated,
                      xOperation,xInformed,xBooked,
                      xAdmMorning,xAdmAfternoon,xAdmEvening>@(3of3),
                 j:$journal<true,xNrOne,xNrTwo,
                      xOperation,xInformed,xBooked,
                      xAdmMorning,xAdmAfternoon,xAdmEvening>@(1of2))=
      def Rounds<xNrOne:Bool,xNrTwo:Bool,xDischarge:Bool, // {{{
                 xJournalUpdated:Bool,
                 xOperation:Bool,xInformed:Bool,xBooked:Bool,
                 xAdmMorning:Bool,xAdmAfternoon:Bool,xAdmEvening:Bool>
                 (b:$treatment_2114_stuegang<xNrOne,xNrTwo,xDischarge,
                          true,xJournalUpdated,
                          xOperation,xInformed,xBooked,
                          xAdmMorning,xAdmAfternoon,xAdmEvening>@(3of3)) =
        guivalue(3,b,3,"NrOne",if xNrOne then "Yes" else "No");
        guivalue(3,b,3,"NrTwo",if xNrTwo then "Yes" else "No");
        guivalue(3,b,3,"Discharged",if xDischarge then "Yes" else "No");
        guivalue(3,b,3,"Journal is here","Yes");
        guivalue(3,b,3,"Journal has changes",if xJournalUpdated then "Yes" else "No");
        guivalue(3,b,3,"Surgeryes",if xOperation then "Yes" else "No");
        guivalue(3,b,3,"Patient is informed",if xInformed then "Yes" else "No");
```

```
  guivalue(3,b,3,"OR is booked",if xBooked then "Yes" else "No");
  guivalue(3,b,3,"Administer in the Morning",if xAdmMorning then "Yes" else "No
      ");
  guivalue(3,b,3,"Administer in the Afternoon",if xAdmAfternoon then "Yes" else
      "No");
  guivalue(3,b,3,"Administer in the Evening",if xAdmEvening then "Yes" else "No
      ");
  guisync(3,b,3)
  {#Surgery():
    b[3]>>operer;
    b[3]>>patientinformeret;
    Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,operer,patientinformeret,
        xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
   #Discharge[[xNrOne and xNrTwo]]():
    b[3]>>udskriv;
    Rounds<xNrOne,xNrTwo,udskriv,xJournalUpdated,xOperation,xInformed,xBooked,
        xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
   #Medication():
    b[3]>>medicin;
    b[3]>>morgen;
    b[3]>>eftermiddag;
    b[3]>>aften;
    Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,xBooked
        ,morgen,eftermiddag,aften>(b),
   ^StopRounds(kommentar:String=""):
    b[3]>>j;
    WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
        xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j)
  } // }}}
in
def WithoutJournal<xNrOne:Bool,xNrTwo:Bool,xDischarge:Bool, // {{{
              xJournalUpdated:Bool,
              xOperation:Bool,xInformed:Bool,xBooked:Bool,
              xAdmMorning:Bool,xAdmAfternoon:Bool,xAdmEvening:Bool>
        (b:$treatment_2114<xNrOne,xNrTwo,xDischarge,
                 false,xJournalUpdated,
                 xOperation,xInformed,xBooked,
                 xAdmMorning,xAdmAfternoon,xAdmEvening>@(3of3),
         s:$fetch_journal<xNrOne,xNrTwo,xOperation,xInformed,xBooked,xAdmMorning
             ,xAdmAfternoon,xAdmEvening>@(1of2))=
  guivalue(3,b,3,"NrOne",if xNrOne then "Yes" else "No");
  guivalue(3,b,3,"NrTwo",if xNrTwo then "Yes" else "No");
  guivalue(3,b,3,"Discharged",if xDischarge then "Yes" else "No");
  guivalue(3,b,3,"Journal is here","No");
  guivalue(3,b,3,"Journal has changes",if xJournalUpdated then "Yes" else "No");
  guivalue(3,b,3,"Surgeryes",if xOperation then "Yes" else "No");
  guivalue(3,b,3,"Patient is informed",if xInformed then "Yes" else "No");
  guivalue(3,b,3,"OR is booked",if xBooked then "Yes" else "No");
  guivalue(3,b,3,"Administer in the Morning",if xAdmMorning then "Yes" else "No
      ");
```

```
        guivalue(3,b,3,"Administer in the Afternoon",if xAdmAfternoon then "Yes" else
            "No");
        guivalue(3,b,3,"Administer in the Evening",if xAdmEvening then "Yes" else "No
            ");
        guisync(3,b,3)
        {^FetchJournal(kommentar:String=""):
          s[2]>>j;
          WithJournal<xNrOne,xNrTwo,xDischarge,false,xOperation,xInformed,xBooked,
              xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j)
        }
//        WithoutJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed
    ,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j) // }}}
       in
        guivalue(3,b,3,"NrOne",if xNrOne then "Yes" else "No");
        guivalue(3,b,3,"NrTwo",if xNrTwo then "Yes" else "No");
        guivalue(3,b,3,"Discharged",if xDischarge then "Yes" else "No");
        guivalue(3,b,3,"Journal is here","Yes");
        guivalue(3,b,3,"Journal has changes",if xJournalUpdated then "Yes" else "No");
        guivalue(3,b,3,"Surgeryes",if xOperation then "Yes" else "No");
        guivalue(3,b,3,"Patient is informed",if xInformed then "Yes" else "No");
        guivalue(3,b,3,"OR is booked",if xBooked then "Yes" else "No");
        guivalue(3,b,3,"Administer in the Morning",if xAdmMorning then "Yes" else "No
            ");
        guivalue(3,b,3,"Administer in the Afternoon",if xAdmAfternoon then "Yes" else
            "No");
        guivalue(3,b,3,"Administer in the Evening",if xAdmEvening then "Yes" else "No
            ");
        guisync(3,b,3)
        {^Rounds[[not xAdmMorning and not xAdmAfternoon and not xAdmEvening]](
            kommentar:String=""):
          b[2]<<j;
          Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,xBooked
              ,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
         #Dictate():
          WithJournal<xNrOne,xNrTwo,xDischarge,true,xOperation,xInformed,xBooked,
              xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
         ^SendJournal[[xJournalUpdated]](kommentar:String=""):
          link(2,secretary,s,1);
          s[1]<<xNrOne;
          s[1]<<xNrTwo;
          s[1]<<xOperation;
          s[1]<<xInformed;
          s[1]<<xBooked;
          s[1]<<xAdmMorning;
          s[1]<<xAdmAfternoon;
          s[1]<<xAdmEvening;
          s[1]<<j;
          WithoutJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed
              ,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,s),
         #Move(room:String=""): //FIXME: Brug nuvaerende room som default
          j[1]<<^WriteRoom;
```

```
 j[1]<<room;
 WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
     xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
^VisitMorning(kommentar:String=""):
 b[3]>>xNrOne;
 b[2]<<xNrOne;
 b[3]>>xNrTwo;
 b[2]<<xNrTwo;
 j[1]<<^WriteNrOne;
 j[1]<<xNrOne;
 j[1]<<^WriteNrTwo;
 j[1]<<xNrTwo;
 WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
     xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
^VisitAfternoon(kommentar:String=""):
 b[3]>>xNrOne;
 b[2]<<xNrOne;
 b[3]>>xNrTwo;
 b[2]<<xNrTwo;
 j[1]<<^WriteNrOne;
 j[1]<<xNrOne;
 j[1]<<^WriteNrTwo;
 j[1]<<xNrTwo;
 WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
     xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
^VisitEvening(kommentar:String=""):
 b[3]>>xNrOne;
 b[2]<<xNrOne;
 b[3]>>xNrTwo;
 b[2]<<xNrTwo;
 j[1]<<^WriteNrOne;
 j[1]<<xNrOne;
 j[1]<<^WriteNrTwo;
 j[1]<<xNrTwo;
 WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
     xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
^AdmMorning[[xAdmMorning]](kommentar:String=""):
 j[1]<<^WriteAdministration;
 j[1]<<false;
 j[1]<<xAdmAfternoon;
 j[1]<<xAdmEvening;
WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
    xBooked,false,xAdmAfternoon,xAdmEvening>(b,j),
^AdmAfternoon[[xAdmAfternoon]](kommentar:String=""):
 j[1]<<^WriteAdministration;
 j[1]<<xAdmMorning;
 j[1]<<false;
 j[1]<<xAdmEvening;
 WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
     xBooked,xAdmMorning,false,xAdmEvening>(b,j),
^AdmEvening[[xAdmEvening]](kommentar:String=""):
```

```
713           j[1]<<^WriteAdministration;
              j[1]<<xAdmMorning;
              j[1]<<xAdmAfternoon;
              j[1]<<false;
              WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
                  xBooked,xAdmMorning,xAdmAfternoon,false>(b,j),
            ^Book[[xOperation and not xBooked]](kommentar:String=""):
              j[1]<<^WriteBooked;
              j[1]<<true;
              WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,
                  true,xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j),
            ^Surgery[[xOperation and xInformed and xBooked]](kommentar:String=""):
723           guisync(3,b,3)
              {^StopSurgery(kommentar:String=kommentar):
                j[1]<<^WriteOperation;
                j[1]<<false;
                j[1]<<^WriteInformed;
                j[1]<<false;
                j[1]<<^WriteBooked;
                j[1]<<false;
                WithJournal<xNrOne,xNrTwo,xDischarge,xJournalUpdated,false,false,false,
                    xAdmMorning,xAdmAfternoon,xAdmEvening>(b,j)
              },
733           ^Discharge[[xDischarge]](kommentar:String=""):
                j[1]<<^WriteEnrolled;
                j[1]<<false;
                j[1]<<^Archive;
                end
            } // }}}
          in
            s[3]>>j; // Modtag journal
            WithJournal<false,false,false,false,false,false,false,false,false,false>(s,j)
      )
743   in Nurse() | // }}}
  // }}}
  // Reception {{{
  // Reception Interface (sReception) {{{
  // Participants:
  // 1: Patient
  // 2: Receptionist
  define $sReception =
    1=>2:2<String>; // Send Name
    1=>2:2<String>; // Send CPR
753 1=>2:2<String>; // Send Symptoms
    {^Enroll:
      2=>1:1<$treatment_2114<false,false,false,true,false,false,false,false,false,false,
          false>@(1of3)>;
      Gend
    }
  in // }}}
  //// Create Channel: sReception {{{
```

```
    (nu sReception : $sReception) // }}}
    ( // Receptionist {{{
      def Receptionist() =
763     link(2,sReception,s,2);
        s[2]>>name;
        s[2]>>cpr;
        s[2]>>symptoms;
        guivalue(2,s,2,"Name:",name);
        guivalue(2,s,2,"CPR:",cpr);
        guivalue(2,s,2,"Symptoms:",symptoms);
        guisync(2,s,2)
        {^Enroll(room:String="Room12a"): // Input room from UI
          link(2,journal,j,1); // Create Journal
773       j[1]<<name;
          j[1]<<cpr;
          j[1]<<room;
          j[1]<<symptoms;
          link(3,treatment,b,1); // Create Treatment session
          b[3]<<j; // Send journal to nurse
          s[1]<<b; // Send treatment session to patient
          Receptionist()
        }
      in Receptionist() | // Create one sReceptionist }}}
783 // }}}
    ( def Patient(name: String,cpr: String,symptoms: String) = // {{{
        def Treatment<xNrOne:Bool,xNrTwo:Bool,xDischarge:Bool,
                      xJournalHere:Bool,xJournalUpdated:Bool,
                      xOperation:Bool,xInformed:Bool,xBooked:Bool,
                      xAdmMorning:Bool,xAdmAfternoon:Bool,xAdmEvening:Bool>
              (b:$treatment_2114<xNrOne,xNrTwo,xDischarge,
                        xJournalHere,xJournalUpdated,
                        xOperation,xInformed,xBooked,
                        xAdmMorning,xAdmAfternoon,xAdmEvening>@(1of3))=
793       def Rounds<xNrOne:Bool,xNrTwo:Bool,xDischarge:Bool, // {{{
                      xJournalUpdated:Bool,
                      xOperation:Bool,xInformed:Bool,xBooked:Bool,
                      xAdmMorning:Bool,xAdmAfternoon:Bool,xAdmEvening:Bool>
                      (b:$treatment_2114_stuegang<xNrOne,xNrTwo,xDischarge,
                               true,xJournalUpdated,
                               xOperation,xInformed,xBooked,
                               xAdmMorning,xAdmAfternoon,xAdmEvening>@(1of3)) =
          guivalue(3,b,1,"NrOne",if xNrOne then "Yes" else "No");
          guivalue(3,b,1,"NrTwo",if xNrTwo then "Yes" else "No");
803       guivalue(3,b,1,"Discharged",if xDischarge then "Yes" else "No");
          guivalue(3,b,1,"Journal has changes",if xJournalUpdated then "Yes" else "No");
          guivalue(3,b,1,"Surgeryes",if xOperation then "Yes" else "No");
          guivalue(3,b,1,"Patient is informed",if xInformed then "Yes" else "No");
          guivalue(3,b,1,"OR is booked",if xBooked then "Yes" else "No");
          guivalue(3,b,1,"Administer in the Morning",if xAdmMorning then "Yes" else "No
              ");
```

```
  guivalue(3,b,1,"Administer in the Afternoon",if xAdmAfternoon then "Yes" else
      "No");
  guivalue(3,b,1,"Administer in the Evening",if xAdmEvening then "Yes" else "No
      ");
  guisync(3,b,1)
  {#Surgery():
    b[1]>>operer;
    b[1]>>patientinformeret;
    Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,operer,patientinformeret,
        xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
   #Discharge[[xNrOne and xNrTwo]]():
    b[1]>>udskriv;
    Rounds<xNrOne,xNrTwo,udskriv,xJournalUpdated,xOperation,xInformed,xBooked,
        xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
   #Medication():
    b[1]>>medicin;
    guivalue(3,b,1,"Medication",medicin);
    b[1]>>xAdmMorning;
    b[1]>>xAdmAfternoon;
    b[1]>>xAdmEvening;
    Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,xBooked
        ,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
   ^StopRounds():
    Treatment<xNrOne,xNrTwo,xDischarge,true,xJournalUpdated,xOperation,xInformed
        ,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b)
  } // }}}
in
  guivalue(3,b,1,"NrOne",if xNrOne then "Yes" else "No");
  guivalue(3,b,1,"NrTwo",if xNrTwo then "Yes" else "No");
  guivalue(3,b,1,"Discharged",if xDischarge then "Yes" else "No");
  guivalue(3,b,1,"Journal is here",if xJournalHere then "Yes" else "No");
  guivalue(3,b,1,"Journal has changes",if xJournalUpdated then "Yes" else "No");
  guivalue(3,b,1,"Surgeryes",if xOperation then "Yes" else "No");
  guivalue(3,b,1,"Patient is informed",if xInformed then "Yes" else "No");
  guivalue(3,b,1,"OR is booked",if xBooked then "Yes" else "No");
  guivalue(3,b,1,"Administer in the Morning",if xAdmMorning then "Yes" else "No
      ");
  guivalue(3,b,1,"Administer in the Afternoon",if xAdmAfternoon then "Yes" else
      "No");
  guivalue(3,b,1,"Administer in the Evening",if xAdmEvening then "Yes" else "No
      ");
  guisync(3,b,1)
  {^Rounds[[xJournalHere and not xAdmMorning and not xAdmAfternoon and not
      xAdmEvening]]():
    Rounds<xNrOne,xNrTwo,xDischarge,xJournalUpdated,xOperation,xInformed,xBooked
        ,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
   #Dictate[[xJournalHere]]():
    Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,true,xOperation,xInformed,
        xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
   ^SendJournal[[xJournalHere and xJournalUpdated]]():
```

```
        Treatment<xNrOne,xNrTwo,xDischarge,false,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
       ^FetchJournal[[not xJournalHere]]():
        Treatment<xNrOne,xNrTwo,xDischarge,true,false,xOperation,xInformed,xBooked,
            xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
       #Move[[xJournalHere]]():
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
       ^VisitMorning[[xJournalHere]](xNrOne:Bool=xNrOne,xNrTwo:Bool=xNrTwo):
        b[3]<<xNrOne;
        b[3]<<xNrTwo;
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
       ^VisitAfternoon[[xJournalHere]](xNrOne:Bool=xNrOne,xNrTwo:Bool=xNrTwo):
        b[3]<<xNrOne;
        b[3]<<xNrTwo;
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
       ^VisitEvening[[xJournalHere]](xNrOne:Bool=xNrOne,xNrTwo:Bool=xNrTwo):
        b[3]<<xNrOne;
        b[3]<<xNrTwo;
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
       ^AdmMorning[[xJournalHere and xAdmMorning]]():
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,false,xAdmAfternoon,xAdmEvening>(b),
       ^AdmAfternoon[[xJournalHere and xAdmAfternoon]]():
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,false,xAdmEvening>(b),
       ^AdmEvening[[xJournalHere and xAdmEvening]]():
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,xBooked,xAdmMorning,xAdmAfternoon,false>(b),
       ^Book[[xJournalHere and xOperation and not xBooked]]():
        Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,xOperation,
            xInformed,true,xAdmMorning,xAdmAfternoon,xAdmEvening>(b),
       ^Surgery[[xJournalHere and xOperation and xInformed and xBooked]](kommentar:
           String=""):
        sync(3,b)
        {^StopSurgery:
          Treatment<xNrOne,xNrTwo,xDischarge,xJournalHere,xJournalUpdated,false,
              false,false,xAdmMorning,xAdmAfternoon,xAdmEvening>(b)
        },
       ^Discharge[[xDischarge and xJournalHere]](): end
      }
in
  link(2,sReception,s,1); // Connect as Patient
  s[2]<<name;
  s[2]<<cpr;
  s[2]<<symptoms;
  sync(2,s)
  {^Enroll:
```

```
        s[1]>>b; // Receive treatment session
        Treatment<false,false,false,true,false,false,false,false,false,false,false>(b)
      }
  in // }}}
// Create Patient Process(es)
( Patient("John Doe","123456....","Congestion")
) ) ) ) ) ) )
```

B

BIT-CODED REGULAR EXPRESSION PARSING

B.1 TRANSDUCER REDUCTION

B.1.1 *Proof of transducer states lower bound (Lemma 59)*

To prove this we define an injective map from $\mathcal{S}[\![T]\!]$ to Q. The intuition of this map should be that each semantic state L is mapped to a node q_L where the paths from q_L to a $q_f \in Q_f$ describes exactly the elements in L.

Consider an $L \in \mathcal{S}[\![T]\!]$. The definition of $\mathcal{S}[\![T]\!]$ gives us three possibilities.

If $L = \mathcal{L}_{oi}[\![T]\!]$ then we map L to $q_L = q_i$.

Otherwise $L \neq \{\}$.

If $L = \mathcal{L}_{oi}[\![T]\!] \downarrow s_L$ for some word $s_L \in \mathcal{L}_{oi}[\![T]\!]$ then $L = \{[]\}$ because T is output deterministic, and we map L to $q_L = q_f$ for some $q_f \in Q_f$ (notice $Q_f \neq \varnothing$ as $\mathcal{L}_{oi}[\![T]\!]$ is not empty).

Finally, if $L = \mathcal{L}_{oi}[\![T]\!] \downarrow s$ where $s@(o :: s') \in \mathcal{L}_{oi}[\![T]\!]$ then there is a path p in T from q_i to some $q_f \in Q_f$ with $\mathrm{tr}_{oi}(p) = s@(o :: s')$. Since o is an output symbol then p can be split into p_1 from q_i to some q_L, and p_2 from q_L to q_f such that $\mathrm{tr}_{oi}(p_1) = s$ and $\mathrm{tr}_{oi}(p_2) = o :: s'$, and we map L to q_L.

This defines a map from $\mathcal{S}[\![T]\!]$ to Q by $L \mapsto q_L$. We need to argue that this map is injective. To do this we consider $T_L = (Q, \Sigma_i, \Sigma_o, q_L, Q_f, t)$ for each $L \in \mathcal{S}[\![T]\!]$. If $L1 \neq L2$ in $\mathcal{S}[\![T]\!]$ then $\mathcal{L}_{oi}[\![T_{L1}]\!] = L1 \neq L2 = \mathcal{L}_{oi}[\![T_{L2}]\!]$ and therefore $T_{L1} \neq T_{L2}$ and this can only mean $q_{L1} \neq q_{L2}$ concluding that the map $L \mapsto q_L$ is injective and thus $|Q| \geqslant |\mathcal{S}[\![T]\!]|$.

B.1.2 *Proof of soundness of $\sim$ (Lemma 60)*

Induction on the length of s.

If $s = []$ then $q_1 \in Q_f$. Since $q_1 \sim q_2$ then $q_2 \in Q_f$ otherwise q_1 and q_2 would have been separated in step 4.a and thus $s = [] \in \mathcal{L}_{oi}[\![T^c_{q_2}]\!]$.

If $s = i :: s'$ where $i \in \Sigma_i$ then $q_1 = q_i$ because T^c is compact which means that only q_i has edges without output. This also means that the 'ε' column for q_1 is set. Since $q_1 \sim q_2$ then the 'ε' column for q_2 must be set, otherwise q_1 and q_2 would be separated in step 4.b. Therefore $q_2 = q_i$ because only q_i can have the 'ε' column

set. This means that $q_1 = q_i = q_2$ and thus $s \in \mathcal{L}_{oi}[\![T^c_{q_2}]\!]$.

If $s = o :: s'$ where $o \in \Sigma_o$ then there is a path $p = (q_1, is, o, q1') :: p'$ from q_1 to some $q_f \in Q_f$ such that $tr_{oi}(p) = o :: (is@tr_{oi}(p')) = s$. Since $q_1 \sim q_2$ there is $(q_2, is, o, q2') \in t$ such that $q_1' \sim q_2'$, otherwise q_1 and q_2 would have been split in step 4.b because of the 'o' column. Now the induction hypothesis (on $tr_{oi}(p')$ and $q_1' \sim q_2'$) yields that $tr_{oi}(p') \in \mathcal{L}_{oi}[\![T^c_{q_2'}]\!]$ and therefore $s = tr_{oi}(p) = o :: (is@tr_{oi}(p')) \in \mathcal{L}_{oi}[\![T^c_{q_2}]\!]$.

B.1.3 *Proof of completeness of ∼ (Lemma 61)*

Induction on iteration (of step 4.b) when q_1 and q_2 were separated.

Start (separated in step 4.a): If q_1 and q_2 were separated in step 4.a, then either $q_1 \in Q_f$ and $q_2 \notin Q_f$ or $q_1 \notin Q_f$ and $q_2 \in Q_f$. In the first case $[] \in \mathcal{L}_{oi}[\![T^c_{q_1}]\!]$ and $[] \notin \mathcal{L}_{oi}[\![T^c_{q_2}]\!]$ so the languages are not the same. In the second case $[] \notin \mathcal{L}_{oi}[\![T^c_{q_1}]\!]$ and $[] \in \mathcal{L}_{oi}[\![T^c_{q_2}]\!]$ so the languages are not the same.

Step (separated in step 4.b): If q_1 and q_2 were separated in step 4.b, then there is a column $c \in \Sigma_o \cup \{\varepsilon\}$ such that q_1 and q_2 differ in column c.

If q_1 and q_2 differ on column ε then either q_1 or q_2 is the initial node, since only the initial node can have the ε column set. If the ε column is set for q_i then $(q_i, is, \varepsilon, q') \in t$ which means that $(is@s') \in \mathcal{L}_{oi}[\![T^c_{q_1}]\!]$ for some s' (as there are no dead states). Since $is \neq []$ then there is a word in $\mathcal{L}_{oi}[\![T^c_{q_i}]\!]$ which starts with an input character. Since this can only be true for the initial node, the languages for q_1 and q_2 are different.

If column 'o' is set for q_1 but unset for q_2 then $(q_1, is, o, q_1') \in t$ which means that $o :: (is@s') \in \mathcal{L}_{oi}[\![T^c_{q_1}]\!]$ for some s' (as there are no dead states). Since q_2 has no edge with output o then $\mathcal{L}_{oi}[\![T^c_{q_2}]\!]$ contains no strings starting with o and therefore the languages are different.

The case where column 'o' is unset for q_1 but set for q_2 follows by symmetry.

If column 'o' for q_1 is (G_1, is_1) and for q_2 it is (G_2, is_2) then $(q_1, is_1, o, q_1') \in t$ and $(q_2, is_2, o, q_2') \in t$ and either $is_1 \neq is_2$ or q_1' and q_2' were separated in an earlier iteration.

If $is_1 \neq is_2$ then there is some $s' \in \mathcal{L}_{oi}[\![T^c_{q_1'}]\!]$ (since there are no dead states), where s' is either empty (if q_1' is final) or starts with an output symbol (because all edges from nodes with input edges have non-ε output). In the same way for all $s'' \in \mathcal{L}_{oi}[\![T^c_{q_2'}]\!]$ either s'' is empty or starts with an output symbol and thus $o :: (is_1@s') \in \mathcal{L}_{oi}[\![T^c_{q_1}]\!]$ and all strings in $\mathcal{L}_{oi}[\![T^c_{q_2}]\!]$ starting with o is on the forms $o :: (is_2@s'') \neq o :: (is_1@s')$ since $is_1 \neq is_2$, so the languages are different.

If $is_1 = is_2$ and q_1' and q_2' were separated in an earlier iteration then the induction hypothesis yields that $\mathcal{L}_{oi}[\![T^c_{q_1'}]\!] \neq \mathcal{L}_{oi}[\![T^c_{q_2'}]\!]$ and since the strings in $\mathcal{L}_{oi}[\![T^c_{q_1}]\!]$ that starts with o are exactly $\{o :: is_1@s' \mid s' \in \mathcal{L}_{oi}[\![T^c_{q_1'}]\!]\}$ and the strings in $\mathcal{L}_{oi}[\![T^c_{q_2}]\!]$ that starts with o are exactly $\{o :: is_2@s' \mid s' \in \mathcal{L}_{oi}[\![T^c_{q_2'}]\!]\}$ the filtered languages are different, and thus the languages are different.

B.1.4 *Proof of minimality of result (Theorem 62)*

Since merging two nodes with the same language preserves the language, then $\mathcal{L}_{oi}[\![T']\!] = \mathcal{L}_{oi}[\![T]\!]$, and therefore Lemma 59 concludes that T' will have at least $|\mathcal{S}[\![T']\!]| = |\mathcal{S}[\![T]\!]|$ nodes.

Since T' is compact (due to step 2), all edges except possibly edges from the initial node are of the form (q_1, is, o, q_2) where $o \in \Sigma_o$. This means that for all $q \in Q'$ $\mathcal{L}_{oi}[\![T'_q]\!]$ will be in $\mathcal{S}[\![T']\!]$. Because Lemma 61 concludes that all nodes with the same language has been merged, we know that $q_1 \neq q_2 \in Q' \Rightarrow \mathcal{L}_{oi}[\![T'_{q1}]\!] \neq \mathcal{L}_{oi}[\![T'_{q2}]\!]$. In conclusion all nodes in T' have languages in $\mathcal{S}[\![T']\!] = \mathcal{S}[\![T]\!]$ and since different nodes have different languages we can conclude that $|Q'| \leqslant |\mathcal{S}[\![T]\!]|$.

C

MULTIPARTY SYMMETRIC SUM TYPES

C.1 PROCESS CONGRUENCE

The process semantics uses the notion of process equivalence ($\equiv$), and this is included in Figure 62.

Figure 62 Process congruence ($\equiv$)

The relation $\equiv$ is defined as the smallest *congruence* relation satisfying

$$\mathsf{P}|\mathbf{0} \equiv \mathsf{P} \qquad \mathsf{P}|\mathsf{Q} \equiv \mathsf{Q}|\mathsf{P}$$

$$\mathsf{P}|(\mathsf{Q}|\mathsf{R}) \equiv (\mathsf{P}|\mathsf{Q})|\mathsf{R} \qquad (\nu n)\mathsf{P}|\mathsf{Q} \equiv (\nu n)(\mathsf{P}|\mathsf{Q}) \text{ if } n \notin \mathsf{fn}(\mathsf{Q})$$

$$(\nu n n')\mathsf{P} \equiv (\nu n' n)\mathsf{P} \qquad (\nu n)\mathbf{0} \equiv \mathbf{0}$$

$$\mathsf{def}\ \mathsf{D}\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0} \qquad (\nu s_1..s_n)\Pi_i s_i : \varnothing \equiv \mathbf{0}$$

$$\mathsf{def}\ \mathsf{D}\ \mathsf{in}\ (\nu n)\mathsf{P} \equiv (\nu n)\mathsf{def}\ \mathsf{D}\ \mathsf{in}\ \mathsf{P} \text{ if } n \notin \mathsf{fn}(\mathsf{D})$$

$$(\mathsf{def}\ \mathsf{D}\ \mathsf{in}\ \mathsf{P}) \mid \mathsf{Q} \equiv \mathsf{def}\ \mathsf{D}\ \mathsf{in}\ (\mathsf{P} \mid \mathsf{Q}) \text{ if } \mathsf{dpv}(\mathsf{D}) \cap \mathsf{fpv}(\mathsf{Q}) = \varnothing$$

$$\mathsf{def}\ \mathsf{D}\ \mathsf{in}\ \mathsf{def}\ \mathsf{D}'\ \mathsf{in}\ \mathsf{P} \equiv \mathsf{def}\ \mathsf{D}\ \mathsf{and}\ \mathsf{D}'\ \mathsf{in}\ \mathsf{P} \text{ if } \mathsf{dpv}(\mathsf{D}) \cap \mathsf{dpv}(\mathsf{D}') = \varnothing$$

C.2 SYMMETRIC SUM TYPES

We include the full set of typing rules in Figure 43. The typing system uses the notion of type projection ($\upharpoonright$) defined in Figure 64, and the notion of subtyping ($\leqslant$) of session-environments which consists of subtyping of each of the used local types defined in Figure 65.

C.3 SUBJECT REDUCTION

We include the proof of subject reduction in Proof C.3.2. The proof uses Lemma C.3.1 which is an extension of Lemma 5.18 from [61] and states that the [Subs] rules can be propagated upwards to the [Sel] and [Branch] rules.

Lemma C.3.1 (Extension of Lemma 5.18 from [61]: Permutation)**.**

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta'}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta''} \quad \textit{then} \quad \dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta''}$$

(1) If *and the result of $\mathcal{E}\,[\![]\!]$ is unchanged.*

$$\dfrac{\dfrac{[X] \quad \mathcal{D}}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta'}$$

(2) If *and the second last rule-application X is not Sel or Branch then the last two rule-applications can be permuted and the result of $\mathcal{E}\,[\![]\!]$ is unchanged.*

Proof:

(1) Is immediate because $\leqslant_{\text{sub}}$ is transitive and $\mathcal{E}\,[\![]\!]$ in both cases reduces to $\mathcal{E}\,[\![\mathcal{D}]\!]$.

(2) Is proved for each possible rule X. This is done as in the original proof, where preservation of $\mathcal{E}\,[\![]\!]$ is shown by evaluation since $\mathcal{E}\left[\!\!\left[\dfrac{[\textsc{Subs}] \quad \Gamma \vdash P \rhd_{\tilde{t}} \Delta'}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta'} \right]\!\!\right] = \mathcal{E}\,[\![\mathcal{D}']\!]$. There is one new case, and we will prove it now.

Sync: In this case we consider a derivation

$$\dfrac{\dfrac{\dfrac{}{[\textsc{Sync}] \quad \forall l \in L'' \quad \dfrac{\mathcal{D}_l}{\Gamma \vdash P_l \rhd_{\tilde{t}} \Delta, \tilde{s} : \{T_l@(p,n)\}}}{\Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{l : P_l\}_{l \in L''} \rhd_{\tilde{t}} \Delta, \tilde{s} : \{\{l : T_l\}_{l \in L;L'}@(p,n)\}}\,[\textsc{Subs}]}{\Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{l : P_l\}_{l \in L''} \rhd_{\tilde{t}} \Delta', \tilde{s} : \{\{l : T'_l\}_{l \in L;L'}@(p,n)\}}$$

where $T_l \leqslant_{\text{sub}} T'_l$ for each $l \in L''$ and $\Delta \leqslant_{\text{sub}} \Delta'$. We can therefore create

$$\dfrac{\dfrac{\dfrac{}{[\textsc{Sync}] \quad \forall l \in L'' \quad \dfrac{\dfrac{\mathcal{D}_l}{\Gamma \vdash P_l \rhd_{\tilde{t}} \Delta, \tilde{s} : \{T_l@(p,n)\}}\,[\textsc{Subs}]}{\Gamma \vdash P_l \rhd_{\tilde{t}} \Delta', \tilde{s} : \{T'_l@(p,n)\}}}{\Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{l : P_l\}_{l \in L''} \rhd_{\tilde{t}} \Delta', \tilde{s} : \{\{l : T'_l\}_{l \in L;L'}@(p,n)\}}$$

Now we only need to show that $\mathcal{E}\,[\![]\!]$ is the same for both derivations, but this is fulfilled, since

$\mathcal{E}\left[\!\!\left[\dfrac{[\textsc{Subs}] \quad \Gamma \vdash P \rhd_{\tilde{t}} \Delta}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta} \right]\!\!\right] = \mathcal{E}\,[\![\mathcal{D}]\!]$. □

Proof C.3.2 (Theorem 4.3.2: Subject reduction)**.**
We prove

If $\Gamma \vdash P \rhd_{\tilde{s}} \Delta$, Δ *coherent and* $P \to P'$
then $\Gamma \vdash P' \rhd_{\tilde{s}} \Delta'$*where* $\Delta \to^{0/1} \Delta'$.

By induction on the derivation of $P \to P'$.
We only have to consider the case Sync, as the other cases are proved in [61]. Assume

$$\frac{[\textsc{Sync}] \qquad h \in \bigcap_{i=1}^{n} L_i}{sync_{\tilde{s},n}\{l : P_{1l}\}_{l\in L_1} \mid \ldots \mid sync_{\tilde{s},n}\{l : P_{nl}\}_{l\in L_n} \to P_{1h} \mid \ldots \mid P_{nh}}$$

We can assume that the typing $\Gamma \vdash sync_{\tilde{t},n}\{l : P_{1l}\}_{l\in L_1} \mid \ldots \mid sync_{\tilde{t},n}\{l : P_{nl}\}_{l\in L_n} \rhd_{\tilde{s}} \Delta, \tilde{t} : \{\{l : T_{li}\}_{l\in L;L'}@(i,n)\}_{i\in\{1..n\}}$ *starts with* $n-1$ *applications of the Conc rule each containing one application of the Sync rule because of the extension of Lemma 5.18 in C.3.1. This gives us the subderivations:*

$$\frac{[\textsc{Sync}] \quad \dfrac{\mathcal{D}_{il}}{\Gamma \vdash P_{il} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{T_{li}@(i,n)\}} \quad \forall l \in L_i}{\Gamma \vdash \{l : P_{il}\}_{l\in L_i} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{l : T_{li}\}_{l\in L';L}@(i,n)}$$

for i=1..n such that $\tilde{s}_i \cap \tilde{s}_j = \varnothing$ *for all* $i \neq j$ *in* $1..n$, $\bigcup_{i=1}^{n} \tilde{s}_i = \tilde{s}$ *and* $\Delta_1 \circ (\Delta_2 \circ (\ldots \circ \Delta_n)) = \Delta$.

Since each of these subderivations starts with the Sync rule we get that

$$\frac{\mathcal{D}_{ih}}{\Gamma \vdash P_{ih} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{T_{hi}@(i,n)\}} \quad \textit{for } i = 1..n$$

Now we can apply Conc $n-1$ *times to create a derivation of*
$\Gamma \vdash P_{1h} \mid \ldots \mid P_{nh} \rhd_{\tilde{s}} \Delta, \tilde{t} : \{T_{hi}@(p,n)\}_{p\in\{1..n\}}$.
Since $\Delta, \tilde{t} : \{\{l : T_{li}\}_{l\in L;L'}@(i,n)\}_{i\in\{1..n\}} \to \Delta, \tilde{t} : \{T_{hi}@(i,n)\}_{i\in\{1..n\}}$, *subject reduction is fulfilled in the Sync case.* □

C.4 ERASURE DEFINITION

We provide the erasure definition and the translation giving the types of the erased processes. Figure 66 shows the conductor generation from(step 2). Figure 67 shows the type and environment translations giving the types of the erased processes, used to express the theorems.

C.5 TYPE PRESERVATION

We include the proof that the presented erasure preserves typability.

Theorem C.5.2 proves that the generated conductors are well-typed in the translated environment, using Lemma C.5.1. Lemma C.5.3 proves that translating a global type, and projecting the result is the same as projecting the original type and translating the result. Using Theorem C.5.2 and Lemma C.5.3 Proof C.5.4 concludes that the erased processes are well-typed in the translated environments, thus proving Theorem 4.4.1.

Lemma C.5.1 (Conductors are typed). *If* $n \geqslant \mathsf{max}(\mathsf{pid}(G))$ *and* $|\tilde{s}| \geqslant \mathsf{max}(\mathsf{sid}(G))$ *then*

$$\varnothing \vdash \mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n} \rhd \tilde{s}, \mathit{in}_{\tilde{s}1}, \ldots, \mathit{out}_{\tilde{s}n} : (([\![G]\!]^{\star}_{n,|\tilde{s}|}) \upharpoonright (\mathit{n{+}1}))@(\mathit{n{+}1}, n+1)$$

Proof: By structural induction on G. The interesting cases Branch and Sync are explained briefly.

Branch: Assume $G = p_1 \to p_2 : k\{l : G_l\}_{l \in L}$.
Then the resulting conductor process is:

$$\mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n} = \mathsf{out}_{\tilde{s}p_1} \rhd \{l : \mathcal{C}\,[\![G_l]\!]^{\star}_{\tilde{s},n}\}_{l \in L},$$

and the global type is:

$$[\![G]\!]^{\star}_{n,|\tilde{s}|} = p_1 \to p_2 : k\{l : p_1 \to n{+}1 : (|\tilde{s}| + 2 \cdot p_1)\{l : [\![G_l]\!]^{\star}_{n,|\tilde{s}|}\}\}_{l \in L}.$$

Therefore the resulting local type is:

$$\begin{aligned} &([\![G]\!]^{\star}_{n,|\tilde{s}|}) \upharpoonright (n{+}1) \\ &= \mathsf{max}_{\leqslant_{\mathsf{sub}}}\{T' \mid \forall l \in L.T' \leqslant_{\mathsf{sub}} (|\tilde{s}| + 2 \cdot p_1)\&\{l : [\![G_l]\!]^{\star}_{n,|\tilde{s}|} \upharpoonright (n{+}1)\}\} \\ &= (|\tilde{s}| + 2 \cdot p_1)\&\{l : [\![G_l]\!]^{\star}_{n,|\tilde{s}|} \upharpoonright (n{+}1)\}_{l \in L} \text{ with } n{+}1 > p_1 \text{ and } p_2. \end{aligned}$$

By the induction hypothesis,

$$\begin{aligned} &\varnothing \vdash \mathcal{C}\,[\![G_l]\!]^{\star}_{\tilde{s},n} \rhd \tilde{s}, \mathsf{in}_{\tilde{s}1}, \ldots, \mathsf{out}_{\tilde{s}n} : ([\![G_l]\!]^{\star}_{n,|\tilde{s}|} \upharpoonright (n{+}1))@(n{+}1, n+1) \\ &\text{for all } l \in L, \end{aligned}$$

and because of rule Branch we get that

$$\begin{aligned} &\varnothing \vdash \mathsf{out}_{\tilde{s}p_1} \rhd \{l : \mathcal{C}\,[\![G_l]\!]^{\star}_{\tilde{s},n}\}_{l \in L} \rhd \tilde{s}, \mathsf{in}_{\tilde{s}1}, \ldots, \mathsf{out}_{\tilde{s}n} : \\ &((|\tilde{s}| + 2 \cdot p_1)\&\{[\![G_l]\!]^{\star}_{n,|\tilde{s}|} \upharpoonright (n{+}1)\}_{l \in L})@(n{+}1, n+1). \end{aligned}$$

Therefore this case is fulfilled.

Sync: Assume $G = \{l : G_l\}_{l \in L; L'}$. In this case the resulting conductor process is:

$$\begin{aligned} &\mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n} \\ &= \mathsf{out}_{\tilde{s}1} \rhd \{\mathsf{cases}_{L''_1 \cup L'} : \mathsf{out}_{\tilde{s}2} \rhd \{\mathsf{cases}_{L''_2 \cup L'} : \ldots \mathsf{out}_{\tilde{s}n} \rhd \{\mathsf{cases}_{L''_n \cup L'} : \\ &\quad \mathsf{rand}\{l : \mathsf{in}_{\tilde{s}1} \lhd l; \ldots; \mathsf{in}_{\tilde{s}n} \lhd l; \mathcal{C}\,[\![G_l]\!]^{\star}_{\tilde{s},n}\}_{l \in L' \cup (L''_1 \cap L''_2 \cap \ldots L''_n)} \\ &\quad \}_{L''_n \subseteq L} \ldots\}_{L''_2 \subseteq L}\}_{L''_1 \subseteq L}, \end{aligned}$$

the resulting global type is

$$
\begin{aligned}
&[\![G]\!]^{\star}_{n,|\tilde{s}|}\\
=\ &1 \to n{+}1 : (|\tilde{s}|+2)\{\mathsf{cases}_{L_1''\cup L'} :\\
&2 \to n{+}1 : (|\tilde{s}|+4)\{\mathsf{cases}_{L_2''\cup L'} : \ldots\\
&n \to n{+}1 : (|\tilde{s}|+2\cdot n)\{\mathsf{cases}_{L_n''\cup L'} :\\
&n{+}1 \to 1 : (|\tilde{s}|+1)\{l :\\
&n{+}1 \to 2 : (|\tilde{s}|+3)\{l : \ldots\\
&n{+}1 \to n : (|\tilde{s}|+2\cdot n-1)\{l : [\![G_l]\!]^{\star}_{n,|\tilde{s}|}\}\ldots\}\\
&\}_{l\in L'\cup(L_1''\cap L_2''\cap\ldots L_n'')}\}_{L_n''\subseteq L}\cdots\}_{L_2''\subseteq L}\}_{L_1''\subseteq L}
\end{aligned}
$$

and since $n{+}1 > p_1$ and p_2 the resulting local type is

$$
\begin{aligned}
&([\![G]\!]^{\star}_{n,|\tilde{s}|})\upharpoonright(n{+}1)\\
=\ &(|\tilde{s}|+2)\&\{\mathsf{cases}_{L_1''\cup L'} :\\
&(|\tilde{s}|+4)\&\{\mathsf{cases}_{L_2''\cup L'} : \ldots\\
&(|\tilde{s}|+2\cdot n)\&\{\mathsf{cases}_{L_n''\cup L'} :\\
&(|\tilde{s}|+1)\oplus\{l :\\
&(|\tilde{s}|+3)\oplus\{l : \ldots\\
&(|\tilde{s}|+2\cdot n-1)\oplus\{l : [\![G_l]\!]^{\star}_{n,|\tilde{s}|}\upharpoonright(n{+}1)\}\ldots\}\\
&\}_{l\in L'\cup(L_1''\cap L_2''\cap\ldots L_n'')}\}_{L_n''\subseteq L}\cdots\}_{L_2''\subseteq L}\}_{L_1''\subseteq L}
\end{aligned}
$$

We get from the induction hypothesis that

$$
\frac{\mathcal{D}_l}{\varnothing \vdash \mathcal{C}\,[\![G_l]\!]^{\star}_{\tilde{s},n} \rhd \tilde{s}, \mathsf{in}_{\tilde{s}1}, \ldots, \mathsf{out}_{\tilde{s}n} : ([\![G_l]\!]^{\star}_{n,|\tilde{s}|}\upharpoonright(n{+}1))@(n{+}1, n+1)}
$$

for all $l \in L \cup L'$. We can therefore construct the type derivation in by n levels of Branch rules on top of which is a Rand rule containing n levels of the Label rule containing $\mathcal{D}_l$ for the selected branches l.
Therefore this case is fulfilled.
We have now proved the interesting cases therefore the lemma is fulfilled. □

Theorem C.5.2 (Conductors are typed)**.**
We prove

> *If* $\Gamma \vdash a : \langle G\rangle$ *and* $n = \mathsf{max}(\mathsf{pid}(G))$ *and* $|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G))$
> *then* $[\![\Gamma]\!] \vdash \mathcal{C}\,[\![G]\!]_{\tilde{s},n,a} \rhd \varnothing$

Proof: Since $\Gamma \vdash a : \langle G\rangle$ we have $[\![\Gamma]\!] \vdash a : [\![G]\!]$.
Now $[\![G]\!] = [\![G]\!]^{\star}_{\mathsf{max}(\mathsf{pid}(G)),\mathsf{max}(\mathsf{sid}(G))}$ and therefore Lemma C.5.1 gives us that
$\varnothing \vdash \mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n} \rhd \tilde{s}, \mathsf{in}_{\tilde{s}1}, \ldots, \mathsf{out}_{\tilde{s}n} : (([\![G]\!])\upharpoonright(n{+}1))@(n{+}1, n+1)$
because $|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G))$ and $n = \mathsf{max}(\mathsf{pid}(G))$.
Finally we conclude the desired by applying Macc. □

Lemma C.5.3 (Projection and erasure commutes). *If* $p \leqslant n$, $n \geqslant \mathsf{max}(\mathsf{pid}(G))$ *and* $m \geqslant \mathsf{max}(\mathsf{sid}(G))$
then $[\![G]\!]^{\star}_{m,n} \upharpoonright p = [\![G \upharpoonright p]\!]_{m,n,p}$.

Proof: Structural induction on G. The interesting cases Branch and Sync are explained briefly.

Branch: $G = p_0 \to p_1 : k\{l : G_l\}_{l \in L}$
There are three subcases depending on p.
If $p \neq p_0$ and $p \neq p_1$ then

$$
\begin{aligned}
&[\![p_0 \to p_1 : k\{l : G_l\}_{l\in L}]\!]^{\star}_{n,m} \upharpoonright p \\
&= (p_0 \to p_1 : k\{l : p_1 \to n{+}1 : m + 2 \cdot p_1 : \{l : [\![G_l]\!]^{\star}_{n,m}\}\}_{l\in L}) \upharpoonright p \\
&= \max_{\leqslant_{\mathrm{sub}}}\{T \mid T \leqslant_{\mathrm{sub}} p_1 \to n{+}1 : m + 2 \cdot p_1\{l : [\![G_l]\!]^{\star}_{n,m}\} \upharpoonright p \quad \forall l \in L\} \\
&= \max_{\leqslant_{\mathrm{sub}}}\{T \mid T \leqslant_{\mathrm{sub}} \max_{\leqslant_{\mathrm{sub}}}\{T' \mid T' \leqslant_{\mathrm{sub}} [\![G_l]\!]^{\star}_{n,m} \upharpoonright p\} \quad \forall l \in L\} \\
&= \max_{\leqslant_{\mathrm{sub}}}\{T \mid T \leqslant_{\mathrm{sub}} [\![G_l]\!]^{\star}_{n,m} \upharpoonright p \quad \forall l \in L\} \text{ and} \\
&[\![p_0 \to p_1 : k\{l : G_l\}_{l\in L} \upharpoonright p]\!]_{n,m,p} \\
&= \left[\!\!\left[\max_{\leqslant_{\mathrm{sub}}}\{T \mid T \leqslant_{\mathrm{sub}} G_l \upharpoonright p \quad \forall l \in L\}\right]\!\!\right]_{n,m,p} \\
&= \max_{\leqslant_{\mathrm{sub}}}\{T \mid T \leqslant_{\mathrm{sub}} [\![G_l \upharpoonright p]\!]_{n,m,p} \quad \forall l \in L\} \text{ ([\![]\!] is monotonic)}
\end{aligned}
$$

Now the induction hypothesis proves this case.
The cases where $p = p_0$ and $p = p_1$ are proved in the same way, except less rewriting of max expressions are required.
Sync: $G = \{l : G_l\}_{l\in L;L'}$
If this case

$$
\begin{aligned}
&[\![\{l : G_l\}_{l\in L;L'}]\!]^{\star}_{n,m} \upharpoonright p \\
&= 1 \to n{+}1 : (m+2)\{\mathsf{cases}_{L_1 \cup L'} : \ldots \\
&\quad n \to n{+}1 : (m + 2 \cdot n)\{\mathsf{cases}_{L_n \cup L'} : \\
&\quad n{+}1 \to 1 : (m+1)\{l : \\
&\quad n{+}1 \to 2 : (m+3)\{l : \ldots \\
&\quad n{+}1 \to n : (m + 2 \cdot n - 1)\{l : [\![G_l]\!]^{\star}_{n,m}\} \ldots\} \\
&\quad \}_{l\in L' \cup (L_1 \cap \ldots L_n)}\}_{L_n \subseteq L} \ldots\}_{L_1 \subseteq L}) \upharpoonright p \qquad \text{(rewriting max)} \\
&= \max_{\leqslant_{\mathrm{sub}}}\{T \mid T \leqslant_{\mathrm{sub}} (m + 2 \cdot p) \oplus \{\mathsf{cases}_{L_p \cup L'} : (m + 2 \cdot p - 1)\&\{l : \\
&\qquad\qquad [\![G_l]\!]^{\star}_{n,m} \upharpoonright p\}_{l\in L' \cup (L_1 \cap \ldots L_n)}\}_{L_p \subseteq L} \\
&\qquad\qquad \forall L_1, \ldots, L_{p-1}, L_{p+1}, \ldots L_n \subseteq L\} \\
&= (m + 2 \cdot p) \oplus \{\mathsf{cases}_{L_p \cup L'} : (m + 2 \cdot p - 1)\&\{l : [\![G_l]\!]^{\star}_{n,m} \upharpoonright p\}_{l\in L' \cup L_p}\}_{L_p \subseteq L}
\end{aligned}
$$

Now this case follows by the induction hypothesis.
We have now proved the interesting cases therefore the lemma is fulfilled. □

Proof C.5.4 (Theorem 4.4.1: Type preservation).
We Prove

$$\textit{If}\quad \frac{\mathcal{D}}{\Gamma \vdash P \triangleright \Delta}\quad \textit{then}\ [\![\Gamma]\!] \vdash \mathcal{E}\,[\![\mathcal{D}]\!] \triangleright [\![\Delta]\!]$$

By induction on the type derivation $\mathcal{D}$. The interesting cases Mcast and Sync are explained briefly.

Mcast:

$$\mathcal{D} = \frac{\Gamma \vdash a : \langle G \rangle \quad \dfrac{\mathcal{D}_1}{\Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright 1)@(1, n)} \quad |\tilde{s}| = \max(\mathsf{sid}(G)) \quad n = \max(\mathsf{pid}(G))}{\Gamma \vdash \overline{a}[2..n](\tilde{s}).P \triangleright \Delta}$$

From this we obtain from the induction hypothesis on $\mathcal{D}_1$ that

$$\frac{\mathcal{D}_1'}{[\![\Gamma]\!] \vdash \mathcal{E}\,[\![\mathcal{D}_1]\!] \triangleright [\![\Delta]\!], \tilde{s}, \mathit{in}_{\tilde{s}1}, \ldots, \mathit{out}_{\tilde{s},n} : [\![G \upharpoonright 1]\!]_{n,|\tilde{s}|,p}@(1, n+1)}$$

and therefore Lemma C.5.3 gives us that

$$\frac{\mathcal{D}_1'}{[\![\Gamma]\!] \vdash \mathcal{E}\,[\![\mathcal{D}_1]\!] \triangleright [\![\Delta]\!], \tilde{s}, \mathit{in}_{\tilde{s}1}, \ldots, \mathit{out}_{\tilde{s},n} : [\![G]\!]^{\star}_{n,|\tilde{s}|} \upharpoonright 1@(1, n+1)}\ .$$

Now we can create

$$\mathcal{D}' = \frac{[\textsc{Mcast}] \quad [\![\Gamma]\!] \vdash a : \langle [\![G]\!] \rangle \quad \mathcal{D}_1' \quad |\tilde{s}, \mathit{in}_{\tilde{s}1}, \ldots, \mathit{out}_{\tilde{s}n}| = n + m = \max(\mathsf{sid}([\![G]\!])) \quad n = \max(\mathsf{pid}([\![G]\!]))}{[\![\Gamma]\!] \vdash \overline{a}[2..n{+}1](\tilde{s}, \mathit{in}_{\tilde{s}1}, \ldots, \mathit{out}_{\tilde{s}n}).\mathcal{E}\,[\![\mathcal{D}_1]\!] \triangleright [\![\Delta]\!]}\ .$$

Corollary C.5.2 gives us that $\dfrac{\mathcal{D}_2'}{[\![\Gamma]\!] \vdash \mathcal{C}\,[\![G]\!]_{\tilde{s},n,a} \triangleright \varnothing}$ *and therefore we can prove the desired by*

$$\frac{[\textsc{Conc}] \quad \dfrac{\mathcal{D}_2'}{[\![\Gamma]\!] \vdash \mathcal{C}\,[\![G]\!]_{\tilde{s},n,a} \triangleright \varnothing} \quad \dfrac{\mathcal{D}'}{[\![\Gamma]\!] \vdash \overline{a}[2..n{+}1](\tilde{s}, \mathit{in}_{\tilde{s}1}, \ldots, \mathit{out}_{\tilde{s}n}).\mathcal{E}\,[\![\mathcal{D}_1]\!] \triangleright [\![\Delta]\!]}}{[\![\Gamma]\!] \vdash \mathcal{E}\,[\![\mathcal{D}]\!] \triangleright [\![\Delta]\!]}\ .$$

Sync:

$$\mathcal{D} = \frac{[\textsc{Sync}] \quad \dfrac{\mathcal{D}_l}{\Gamma \vdash P_l \triangleright \Delta, \tilde{s} : T_l@(p, n)} \quad \forall l \in L \quad L'' \subseteq L \cup L' \quad L' \subseteq L''}{\Gamma \vdash \{l : P_l\}_{l \in L''} \triangleright \Delta, \tilde{s} : \{l : T_l\}_{l \in L;L'}@(p, n)}$$

If this case we apply the Branch rule to the results of the induction hypothesis, and then applying a Sel rule to that.

We have now proved the interesting cases therefore the theorem is fulfilled. □

C.6 CONGRUENCE PRESERVATION

We include the proof that the presented erasure preserves process congruence.
We start by extending the erasure to runtime processes in Figure 68. Proof C.6.1 concludes soundness, and can be considered as a natural extension of Theorem 5.22(1) from [61]. Lemme C.6.4 proves that each relevant congruence step for the translated process can be matched by a congruence step by the original process, and Proof C.6.5 uses this to conclude completeness. Thus both soundness and completeness of Theorem 4.4.2 are proved.

Proof C.6.1 (Theorem 4.4.2(Soundness): Congruence preservation)**.**
We prove

$$\textit{If } P \equiv Q \textit{ and } \frac{\mathcal{D}_1}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta} \textit{ then there is a derivation } \frac{\mathcal{D}_2}{\Gamma \vdash Q \rhd_{\tilde{t}} \Delta} \textit{ such that } \mathcal{E}\,[\![\mathcal{D}_1]\!] \equiv \mathcal{E}\,[\![\mathcal{D}_2]\!].$$

By rule-induction on $P \equiv Q$. *All the rules are considered in [61], and we get that* $\mathcal{E}\,[\![\mathcal{D}_1]\!] \equiv \mathcal{E}\,[\![\mathcal{D}_2]\!]$ *for the derivations found in the original proof, by moving the conductor processes around and applying the considered equivalence rule.* □

Observation C.6.2.
If $\mathcal{E}\,[\![\Gamma_1 \vdash P_1 \rhd \Delta_1]\!] = \mathcal{E}\,[\![\Gamma_2 \vdash P_2 \rhd \Delta_2]\!]$ *then* $P_1 = P_2$.

Observation C.6.3.
If $\mathcal{E}\,[\![\mathcal{D}_1]\!] \equiv \mathcal{E}\,[\![\mathcal{D}_2]\!]$ *then there is a derivation without using the steps*

- $Q_1|Q_2 \equiv Q_2|Q_1$
- $Q_1|(Q_2|Q_3) \equiv (Q_1|Q_2)|Q_3$
- $(\nu n)Q_1|Q_2 \equiv (\nu n)(Q_1|Q_2)$

where either Q_1, Q_2 *or* Q_3 *are conductor processes.*

This is true because $\mathcal{E}\,[\![]\!]$ only creates a a conductor process from the derivation of a session requesting process $\overline{a}[2..n](\tilde{s}).P$ and in this case it is created in parallel with the session request process $\overline{a}[2..n+1](\tilde{s}')P'$. Therefore if the steps of $\mathcal{E}\,[\![\mathcal{D}_1]\!] \equiv \mathcal{E}\,[\![\mathcal{D}_2]\!]$ at some point separates the conductor process from the session request, it must later join the session request with another conductor process in order to reach $\mathcal{E}\,[\![\mathcal{D}_2]\!]$. For this reason we can safely assume that the steps of $\mathcal{E}\,[\![\mathcal{D}_1]\!] \equiv \mathcal{E}\,[\![\mathcal{D}_2]\!]$ at no point separates the conductor from the session request, since we can create a derivation that fulfils this by removing all the steps that moves or changes the associativity of conductor processes.

Lemma C.6.4 (Congruence single step completeness)**.**
If $\dfrac{\mathcal{D}_1}{\Gamma_1 \vdash P_1 \rhd \Delta_1}$ *and* $\mathcal{E}[\![\mathcal{D}_1]\!] \equiv Q$ *in one step, that does not use the rules*

- $Q_1|Q_2 \equiv Q_2|Q_1$
- $Q_1|(Q_2|Q_3) \equiv (Q_1|Q_2)|Q_3$
- $(\nu n)Q_1|Q_2 \equiv (\nu n)(Q_1|Q_2)$

where either Q_1, Q_2 *or* Q_3 *are conductor processes, then there is* $\dfrac{\mathcal{D}_2}{\Gamma_1 \vdash P_2 \rhd \Delta_1}$ *such that* $\mathcal{E}[\![\mathcal{D}_2]\!] = Q$ *and* $P_1 \equiv P_2$.

PROOF: This is proved for each rule. We consider the most interesting cases here.
Case $Q_1|Q_2 \equiv Q_2|Q_1$:
Only the typing-rules MCAST and CONC results in processes of this form. If $Q_1|Q_2$ is the result of an MCAST rule, then Q_1 is a conductor process, and since we can assume this is not the case, we only have to consider the CONC rule. Therefore we know that

$$\mathcal{D}_1 = \dfrac{\dfrac{\mathcal{D}_{1i}}{\Gamma_1 \vdash P_{1i} \rhd \Delta_{1i}}\,[\text{CONC}]}{\Gamma_1 \vdash P_{11}|P_{12} \rhd \Delta_{11} \circ \Delta_{12}} \text{ and can select } \mathcal{D}_2 = \dfrac{\dfrac{\mathcal{D}_{1i}}{\Gamma_1 \vdash P_{1i} \rhd \Delta_{1i}}\,[\text{CONC}]}{\Gamma_1 \vdash P_{12}|P_{11} \rhd \Delta_{12} \circ \Delta_{11}}.$$

Since $\mathcal{E}[\![\mathcal{D}_2]\!] = \mathcal{E}[\![\mathcal{D}_{12}]\!]|\mathcal{E}[\![\mathcal{D}_{11}]\!] = Q_2|Q_1$ and $P_{11}|P_{12} \equiv P_{12}|P_{11}$.
Cases $Q_1|(Q_2|Q_3) \equiv (Q_1|Q_2)|Q_3$ and $(\nu n)Q_1|Q_2 \equiv (\nu n)(Q_1|Q_2)$ are proved in the same way, and the rest of the rules are straightforward. □

Proof C.6.5 (Theorem 4.4.2(Completeness): Congruence preservation)**.**
We prove

$$\text{If } \dfrac{\mathcal{D}_1}{\Gamma_1 \vdash P_1 \rhd \Delta_1} \text{ and } \dfrac{\mathcal{D}_2}{\Gamma_2 \vdash P_2 \rhd \Delta_2} \text{ and } \mathcal{E}[\![\mathcal{D}_1]\!] \equiv \mathcal{E}[\![\mathcal{D}_2]\!] \text{ then } P_1 \equiv P_2.$$

By induction on the number of steps in the derivation of $\mathcal{E}[\![\mathcal{D}_1]\!] \equiv \mathcal{E}[\![\mathcal{D}_2]\!]$.
If there are no steps, then $\mathcal{E}[\![\mathcal{D}_1]\!] = \mathcal{E}[\![\mathcal{D}_2]\!]$ *and therefore* $P_1 = P_2$.
If $\mathcal{E}[\![\mathcal{D}_1]\!] \equiv \mathcal{E}[\![\mathcal{D}_2]\!]$ *uses at least one step, and the first step is* $\mathcal{E}[\![\mathcal{D}_1]\!] \equiv Q$ *then Observation C.6.3 lets us use Lemma C.6.4 to find a derivation* $\dfrac{\mathcal{D}_3}{\Gamma_1 \vdash P_3 \rhd \Delta_1}$ *such that* $\mathcal{E}[\![\mathcal{D}_3]\!] = Q$ *and* $P_1 \equiv P_3$. *Now we get from the induction hypothesis that* $P_3 \equiv P_2$ *and therefore* $P_1 \equiv P_2$. □

C.7 ERASURE SOUNDNESS

We prove that the erasure is sound. To do this we define PC as the set of possible partial conductors for the open sessions Δ. Proof C.7.2 proves Theorem 4.4.3 (an extension of Theorem 5.22(2) from [61]) which states soundness for a single step, and Corollary C.7.3 generalises this to soundness for multiple steps.

Definition C.7.1 (Partial conductors: PC)**.** *The possible conductors for an environment of partially completed session* $\mathit{PC}(\Delta)$ *is defined below.*

$$\begin{array}{l}\mathit{PC}(\varnothing) = \{0\}\\ \mathit{PC}(\Delta', \tilde{s} : \{T_p@(p, n)\}_{p\in\{1..n\}}) =\\ \bigcup_{P'_C\in \mathit{PC}(\Delta')}\{\mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n}\,|P'_C|\mathit{in}_{\tilde{s}1} : \varnothing|\ldots|\mathit{out}_{\tilde{s}n} : \varnothing \mid G\upharpoonright p = T_p\ \forall p \in \{1..n\}\}\end{array}$$

Proof C.7.2 (Theorem 4.4.3: Erasure soundness)**.**

If $\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta}$, $P \to P'$, Δ *coherent and* $P_C \in \mathit{PC}(\Delta \circ \Delta^{\circ})$

then there is a derivation $\dfrac{\mathcal{D}'}{\Gamma \vdash P' \rhd_{\tilde{t}} \Delta'}$ *and a* $P'_C \in \mathit{PC}(\Delta' \circ \Delta^{\circ})$
such that $\Delta \to^{0/1} \Delta'$ *and* $\mathcal{E}\,[\![\mathcal{D}]\!]\,|P_C \to^{\star} \mathcal{E}\,[\![\mathcal{D}']\!]\,|P'_C$.

The proof is by induction on the derivation of $P \to P'$*. All cases except Sync are covered by the original proof, where* $\mathcal{E}\,[\![\mathcal{D}]\!]\,|P_C \to^{\star} \mathcal{E}\,[\![\mathcal{D}']\!]\,|P'_C$ *can be proved for the found derivation* $\mathcal{D}'$ *by selecting* P'_C *using the same global types as was used to find* P_C*. We will show the case for Link as an example, and prove the new case for Sync.*

Link*:* $\dfrac{}{\vdash \overline{a}[2..n](\tilde{s}).P_1|a[2](\tilde{s}).P_2|\ldots|a[n](\tilde{s}).P_n \to (\nu\tilde{s})(P_1|P_2|\ldots|P_n|s_1 : \varnothing|\ldots|s_m : \varnothing)}$ [Link]

We can assume that the typing derivation $\mathcal{D}$ *starts with* n *applications of the Conc rule without changing* $\mathcal{E}\,[\![\mathcal{D}]\!]$ *because of the extension of Lemma 5.18 in C.3.1. The first Conc rule contains a derivation*

$$\dfrac{\Gamma \vdash a : \langle G\rangle \quad \dfrac{\mathcal{D}_1}{\Gamma \vdash P_1 \rhd \Delta, \tilde{s} : (G\upharpoonright 1)@(1, n)} \quad |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \quad n = \mathsf{max}(\mathsf{pid}(G))}{\Gamma \vdash \overline{a}[2..n](\tilde{s}).P_1 \rhd \Delta_1}\ [\textsc{Mcast}]$$

and the other Conc rules contain the derivations

$$\dfrac{\Gamma \vdash a : \langle G\rangle \quad \dfrac{\mathcal{D}_p}{\Gamma \vdash P \rhd \Delta_p, \tilde{s} : (G\upharpoonright p)@(p, n)} \quad |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \quad n = \mathsf{max}(\mathsf{pid}(G))}{\Gamma \vdash a[p](\tilde{s}).P_p \rhd \Delta_p}\ [\textsc{Macc}]$$

for $p = 1..n$.

We can now create $\mathcal{D}'$ *as an application of the CRes rule containing* $2 \cdot n$ *applications of the Conc rule containing* $\mathcal{D}_i$ *for* $i = 1..n$*, and* $\mathcal{D}_{n+1}\ldots\mathcal{D}_{2\cdot n}$ *which are derivations for the empty queues.* $\mathcal{D}'$ *concludes that*

$$\Gamma \vdash (\nu\tilde{s})(P_1|P_2|\ldots|P_n|s_1 : \varnothing|\ldots|s_n : \varnothing) \rhd \Delta.$$

Now can now use the definition of $\mathcal{E}\,[\![]\!]$ to find

$$\begin{aligned}\mathcal{E}\,[\![\mathcal{D}]\!] &= a[n{+}1](\tilde{s}, c_{\tilde{s}1}, \ldots, c_{\tilde{s}n}).\mathcal{C}\,[\![G]\!]^{\star}_{\tilde{s},n}\,|\overline{a}[2\,..\,n{+}1](\tilde{s}).\mathcal{E}\,[\![\mathcal{D}_1]\!]\,|a[2](\tilde{s}).\mathcal{E}\,[\![\mathcal{D}_2]\!] \\ &\quad |\ldots|a[n](\tilde{s}).\mathcal{E}\,[\![\mathcal{D}_n]\!] \\ \mathcal{E}\,[\![\mathcal{D}']\!] &= (\nu\tilde{s}, c_{\tilde{s}1}, \ldots, c_{\tilde{s}n})(\mathcal{C}\,[\![G]\!]_{\tilde{s},n,a} \mid c_{\tilde{s}1} : \varnothing \mid \ldots \mid c_{\tilde{s}n} : \varnothing \mid \mathcal{E}\,[\![\mathcal{D}_1]\!]\,|\ldots|\mathcal{E}\,[\![\mathcal{D}_n]\!] \\ &\quad |s_1 : \varnothing|\ldots|s_n : \varnothing)\end{aligned}$$

With these choices of $\mathcal{D}$, $\mathcal{D}'$ the theorem is proved by a single Link step wrapped in a Conc and Str rule since we can choose $P_C = P'_C$.

$$\text{[Sync]}\quad \frac{}{l' \in \bigcap_{i=1}^{n} L_i}$$

Sync: $\mathit{sync}_{\tilde{s},n}\{l : P_{1l}\}_{l\in L_1} \mid \ldots \mid \mathit{sync}_{\tilde{s},n}\{l : P_{nl}\}_{l\in L_n} \to P_{1l'} \mid \ldots \mid P_{nl'}$

We can assume that the typing derivation $\mathcal{D}$ starts with n applications of the Conc rule without changing $\mathcal{E}\,[\![\mathcal{D}]\!]$ because of the extension of Lemma 5.18. The Conc rules contain the derivations

$$\text{[Sync]}\quad \frac{\forall l \in L'' :\ \dfrac{\mathcal{D}_{pl}}{\Gamma \vdash P_{pl} \rhd \Delta_p, \tilde{s} : \{T_{pl}@(p,n)\}} \qquad L'' \subseteq L \cup L' \quad L' \subseteq L''}{\Gamma \vdash \mathit{sync}_{\tilde{s},n}\{l : P_{pl}\}_{l\in L''} \rhd \Delta_p, \tilde{s} : \{\{l : T_{pl}\}_{l\in L;L'}@(p,n)\}} \quad \textit{for } p = 1..n.$$

We can now create $\mathcal{D}'$ as n applications of the Conc rule containing $\mathcal{D}_{1l'}, \ldots, \mathcal{D}_{nl'}$. $\mathcal{D}'$ concludes that $\Gamma \vdash P_{1l'} \mid \ldots \mid P_{nl'} \rhd \Delta_1, \tilde{s} : \{T_{pl'}@(p,n)\}_{p\in\{1..n\}}$.

Let $P_C \in \mathit{PC}(\Delta) = \mathit{PC}(\Delta_1, \tilde{s} : \{\{l : T_{pl}\}_{l\in L;L'}@(p,n)\}_{p\in\{1..n\}})$
$= \bigcup_{P_{C1}\in\mathit{PC}(\Delta_1)}\{\mathcal{C}\,[\![\{l : G_l\}_{l\in L;L'}]\!]^{\star}_{\tilde{s},n}\,|P_{C1}|c_{\tilde{s}1} : \varnothing|\ldots|c_{\tilde{s}n} : \varnothing \mid G_l \upharpoonright p = T_{pl}\ \forall p \in \{1..n\}, l \in L \cup L'\}$.

We can now choose $P'_C = \mathcal{C}\,[\![G_l]\!]^{\star}_{\tilde{s},n}\,|P_{C1}|c_{\tilde{s}1} : \varnothing|\ldots|c_{\tilde{s}n} : \varnothing$.

With these choices of $\mathcal{D}$, $\mathcal{D}'$, P_C and P'_C the theorem is proved by performing the communication which $\mathcal{E}\,[\![]\!]$ and PC produces from the synchronisation constructor. □

Corollary C.7.3 (Erasure soundness)**.**
This corollary is an extension of Theorem 5.22(3) from [61].

We prove

If $\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd \varnothing}$ *and* $P \to^{\star} P'$

then there is a derivation $\dfrac{\mathcal{D}'}{\Gamma \vdash P' \rhd \varnothing}$
such that $\mathcal{E}\,[\![\mathcal{D}]\!] \to^{\star} \mathcal{E}\,[\![\mathcal{D}']\!]$.

Proof: By induction on the number of steps in $P \to^{\star} P'$.
If $P = P'$ then the theorem is trivially fulfilled.

If $P \to^\star P'$ is of the form $P \to P_1 \to^\star P'$ then the extension of Theorem 5.22(2) in Theorem 4.4.3 gives us that there is a derivation $\dfrac{\mathcal{D}_1}{\Gamma \vdash P_1 \rhd \varnothing}$ such that $\mathcal{E}\,[\![\mathcal{D}]\!]\,|0 \to^\star \mathcal{E}\,[\![\mathcal{D}_1]\!]\,|0$, since $\varnothing$ is coherent and complete and $\mathtt{PC}(\varnothing) = \{0\}$. Now we can wrap the first and the last step in $\mathcal{E}\,[\![\mathcal{D}]\!]\,|0 \to^\star \mathcal{E}\,[\![\mathcal{D}_1]\!]\,|0$ with a Str rule to get $\mathcal{E}\,[\![\mathcal{D}]\!] \to^\star \mathcal{E}\,[\![\mathcal{D}_1]\!]$.

The induction hypothesis yields that there is a derivation $\dfrac{\mathcal{D}'}{\Gamma \vdash P' \rhd \varnothing}$ such that $\mathcal{E}\,[\![\mathcal{D}_1]\!] \to^\star \mathcal{E}\,[\![\mathcal{D}']\!]$, and therefore we get that $\mathcal{E}\,[\![\mathcal{D}]\!] \to^\star \mathcal{E}\,[\![\mathcal{D}']\!]$ by combining the steps in the two evaluations.

Therefore the theorem is fulfilled. □

C.8 ERASURE COMPLETENESS

We prove that the erasure is complete. This is done by first classifying all stepping derivations by the active stepping rule in Lemma C.8.1. This classification is used to formally define conduction steps ($\rightharpoonup$) in Definition C.8.2. Next we prove confluence for the conduction steps in Lemma C.8.7, and using this it is possible to prove the single step completeness in Lemma C.8.11. Finally we can conclude multistep completeness in Proof C.8.12 using a non-trivial combination of the single step completeness (Illustrated in Figure 69), and the confluence results, thus proving Theorem 4.4.4.

Step 1: Conduction Steps

Lemma C.8.1 (Classification of steps)**.**
If $P_1 \to P_2$ *then exactly one of the following cases is fulfilled.*

Link $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (\overline{a}[2\ldots n](\tilde{s}).Q_{11}|a[2](\tilde{s}).Q_{12}|\ldots|a[n](\tilde{s}).Q_{1n}|Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ ((\nu\tilde{s})(Q_{11}|Q_{12}|\ldots|Q_{1n}|s_1 : \varnothing|\ldots|s_m : \varnothing)|Q_2)$.

Send $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k!\langle\tilde{e}\rangle; Q_1|s_k : \tilde{h}|Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1|s_k : \tilde{h}\cdot\tilde{v}|Q_2)$ *where* $\tilde{e} \downarrow \tilde{v}$.

Recv $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k?(\tilde{x}); Q_1|s_k : \tilde{v}\cdot\tilde{h}|Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1[\tilde{v}/\tilde{x}]|s_k : \tilde{h}|Q_2)$.

Label $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1|s_k : \tilde{h}|Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1|s_k : \tilde{h}\cdot l|Q_2)$.

Branch $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1, \ldots\}|s_k : l\cdot\tilde{h}|Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1|s_k : \tilde{h}|Q_2)$.

Deleg $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k!\langle\!\langle\tilde{t}\rangle\!\rangle; Q_1 | s_k : \tilde{h} | Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1 | s_k : \tilde{h} \cdot \tilde{t} | Q_2)$.

SRec $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k?((\tilde{t})); Q_1 | s_k : \tilde{t} \cdot \tilde{h} | Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1 | s_k : \tilde{h} | Q_2)$.

IfT $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (\mathsf{if}\ e\ \mathsf{then}\ Q_{11}\ \mathsf{else}\ Q_{12} | Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_{11} | Q_2)$ *where* $e \downarrow \mathsf{true}$.

IfF $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (\mathsf{if}\ e\ \mathsf{then}\ Q_{11}\ \mathsf{else}\ Q_{12} | Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_{12} | Q_2)$ *where* $e \downarrow \mathsf{false}$.

Def $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (X\langle\tilde{e}\tilde{s}\rangle | Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1[\tilde{v}/\tilde{x}] | Q_2)$ *where* $X(\tilde{x}\tilde{s}) = Q_1 \in D$ *and* $\tilde{e} \downarrow \tilde{v}$.

Rand $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (\mathit{rand}\{P_i\}_{i\in I} | Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (P_j | Q_2)$ *for some* $j \in I$.

Sync $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (\mathit{sync}_{\tilde{s},n}\{l : Q_{11}, \ldots\} | \ldots | \mathit{sync}_{\tilde{s},n}\{l : Q_{1n}, \ldots\} | Q_2)$ *and*
$P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_{11} | \ldots | Q_{1n} | Q_2)$.

PROOF: By induction on the derivations of $P_1 \to P_2$.
Each of the rules represented results in the case of the same name.
The *Scop* rule adds n to $\tilde{n}$.
The *Par* rule adds the unused process to Q_2.
The *Defin* rule adds the definitions to D, where reaming can be necessary.
Finally the *Str* rule follows directly by the induction hypothesis since $\equiv$ is transitive.
□

Definition C.8.2 (Definition of conduction steps)**.**
If $P_1 \to P_2$ *then we write* $P_1 \rightharpoonup P_2$ *if one of the following cases is fulfilled*

- *Lemma C.8.1 uses case* ***Label*** *or* ***Branch*** *where* s_k *is a conductor channel.*
- *Lemma C.8.1 uses case* ***Def*** *where* X *is a conductor process-variable.*

Otherwise we write that $P_1 \not\rightharpoonup P_2$.

Step 2: Confluence

Lemma C.8.3 (Separation)**.**
If P_1 *is* linear, $P_1 \rightharpoonup P_2$ *and* $P_1 \to P_2'$
then one of the following eight cases is fulfilled.

- $P_2 \equiv P_2'$.

- $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1|Q_2|Q_3)$
  where $Q_1 \rightarrowtail Q_1'$, $Q_2 \to Q_2'$
  and $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1'|Q_2|Q_3)$
  and $P_2' \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1|Q_2'|Q_3)$.

- $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1'|s_k : \tilde{v} \cdot \tilde{h}|s_k?(\tilde{x}); Q_2'|Q_3)$
  where $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1'|s_k : \tilde{v} \cdot \tilde{h} \cdot l|s_k?(\tilde{x}); Q_2'|Q_3)$
  and $P_2' \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1'|s_k : \tilde{h}|Q_2'[\tilde{v}/\tilde{x}]|Q_3)$.

- $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1'|s_k : \tilde{t} \cdot \tilde{h}|s_k?((\tilde{t})); Q_2'|Q_3)$
  where $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1'|s_k : \tilde{t} \cdot \tilde{h} \cdot l|s_k?((\tilde{t})); Q_2'|Q_3)$
  and $P_2' \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1'|s_k : \tilde{h}|Q_2'|Q_3)$.

- $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1'|s_k : l' \cdot \tilde{h}|s_k \rhd \{l' : Q_2', \ldots\}|Q_3)$
  where $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1'|s_k : l' \cdot \tilde{h} \cdot l|s_k \rhd \{l' : Q_2', \ldots\}|Q_3)$
  and $P_2' \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1'|s_k : \tilde{h}|Q_2'|Q_3)$.

- $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1', \ldots\}|s_k : l \cdot \tilde{h}|s_k!\langle\tilde{e}\rangle; Q_2'|Q_3)$
  where $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1'|s_k : \tilde{h}|s_k!\langle\tilde{e}\rangle; Q_2'|Q_3)$
  and $P_2' \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1', \ldots\}|s_k : l \cdot \tilde{h} \cdot \tilde{v}|Q_2'|Q_3)$ *where* $\tilde{e} \downarrow \tilde{v}$.

- $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1', \ldots\}|s_k : l \cdot \tilde{h}|s_k!\langle\!\langle\tilde{t}\rangle\!\rangle; Q_2'|Q_3)$
  where $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1'|s_k : \tilde{h}|s_k!\langle\!\langle\tilde{t}\rangle\!\rangle; Q_2'|Q_3)$
  and $P_2' \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1', \ldots\}|s_k : l \cdot \tilde{h} \cdot \tilde{t}|Q_2'|Q_3)$.

- $P_1 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1', \ldots\}|s_k : l \cdot \tilde{h}|s_k \lhd l'; Q_2'|Q_3)$
  where $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1'|s_k : \tilde{h}|s_k \lhd l'; Q_2'|Q_3)$
  and $P_2' \equiv (\nu\tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1', \ldots\}|s_k : l \cdot \tilde{h} \cdot l'|Q_2'|Q_3)$.

PROOF: Consider each case of $P_1 \rightarrowtail P_2$ and $P_1' \to P_2'$ in Lemma C.8.1. □

Lemma C.8.4 (Diamond property for $\rightarrowtail$)**.**
If P_1 *is* linear, $P_1 \rightarrowtail P_2$ *and* $P_1 \rightarrowtail P_2'$
then $P_2 \equiv P_2'$ *or* $\exists P3$ *such that* $P_2 \rightarrowtail P_3$ *and* $P_2' \rightarrowtail P_3$.

PROOF: Consider each case of Lemma C.8.3. □

Lemma C.8.5 (Diamond property)**.**
If P_1 *is* linear, $P_1 \rightarrowtail P_2$ *and* $P_1 \rightarrowtail P_2'$
then $\exists P3$ *such that* $P_2 \rightarrowtail P_3$ *and* $P_2' \rightarrowtail P_3$.

PROOF: Consider each case of Lemma C.8.3. □

Lemma C.8.6 (Partial confluence)**.**
If P_1 *is* linear, $P_1 \rightarrowtail^\star P_2$ *and* $P_1 \to P_2'$
then $\exists P3$ *such that* $P_2 \to^\star P_3$ *and* $P_2' \rightarrowtail^\star P_3$.

PROOF: By induction on the number of steps in $P_1 \rightarrowtail^\star P_2$, using Lemma C.8.4 or C.8.5 in each step. □

It should be noted that the number of steps is not increased, such that the number of steps in $P_1 \rightarrowtail^\star P_2$ is an upper bound for number of steps in $P_2' \rightarrowtail^\star P_3$ and $P_2 \rightarrow^\star P_3$ in at most one step.

Lemma C.8.7 (confluence)**.**
If P_1 *is* linear, $P_1 \rightarrowtail^\star P_2$ *and* $P_1 \rightarrow^\star P_2'$
then $\exists P3$ *such that* $P_2 \rightarrow^\star P_3$ *and* $P_2' \rightarrowtail^\star P_3$.

PROOF: By induction on the number of steps in the evaluation $P_1 \rightarrow^\star P_2'$, using Lemma C.8.6 in each step. □

It should be noted that the number of steps is not increased, such that the number of steps in $P_1 \rightarrowtail^\star P_2$ is an upper bound for number of steps in $P_2' \rightarrowtail^\star P_3$ and the number of steps in $P_1 \rightarrow^\star P_2'$ is an upper bound for the number of steps in $P_2 \rightarrow^\star P_3$.

Step 3: Single-step completeness

Lemma C.8.8 (Direct step completeness)**.**
If $\mathcal{E}\left[\!\left[\dfrac{\mathcal{D}_1}{\Gamma \vdash P_1 \rhd \Delta_1} \right]\!\right] = Q_1$,
$P_{c1} \in \mathit{PC}(\Delta_1 \circ \Delta^\circ)$ *and* $Q_1 | P_{c1} \rightarrowtail Q_2$
$\dfrac{\mathcal{D}_2}{}$
then $\exists\ \Gamma \vdash P_2 \rhd \Delta_2$ *and* $P_{c2} \in \mathit{PC}(\Delta_2 \circ \Delta^\circ)$ *such that* $P_1 \rightarrow P_2$, $\Delta_1 \rightarrow^{0/1} \Delta_2$ *and* $Q_2 \rightarrowtail^\star \mathcal{E}\left[\!\left[\dfrac{\mathcal{D}_2}{\Gamma \vdash P_2 \rhd \Delta_2} \right]\!\right] \mid P_{c2}$

PROOF: Check all direct steps of all right-hand-sides of $\mathcal{E}[\![]\!]$ and $\mathcal{C}[\![]\!]$, using that channel-queues are copied, and the newly created channels $\mathtt{in}_{\tilde{s}p}$ and $\mathtt{out}_{\tilde{s}p}$ are new names and therefore does not interfere with original channels. □

Lemma C.8.9 (Step commutativity)**.**
If $\mathcal{E}\left[\!\left[\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd \Delta} \right]\!\right] = Q_1$,
$P_c \in \mathit{PC}(\Delta \circ \Delta^\circ)$ *and* $Q_1 | P_c \rightarrowtail^\star Q_2 \rightarrowtail Q_3$
where Lemma C.8.1 on $Q_2 \rightarrowtail Q_3$ *does not use the Rand case in a conductor*
then $Q_1 | P_c \rightarrowtail Q_2' \rightarrowtail^\star Q_3$.

PROOF: By induction on the call-tree of $\mathcal{E}[\![]\!]$ and $\mathcal{C}[\![]\!]$.
First we can see by inspection of all right-hand-sides of $\mathcal{E}[\![]\!]$ and $\mathcal{C}[\![]\!]$ that if $\mathtt{in}_{\tilde{s}p} : h$ is in $Q_1 | P_c$ then $h = \varnothing$.
Second we can see by inspection of all right-hand-sides of $\mathcal{E}[\![]\!]$ and $\mathcal{C}[\![]\!]$ that

writing on any $\mathtt{in}$ channel is preceded by a non-conduction step.
Finally we can see by inspection of all right-hand-sides of $\mathcal{E}[\![]\!]$ and $\mathcal{C}[\![]\!]$ that either the result starts with a non-conduction step, or non-conduction steps are preceded by reading on an $\mathtt{in}$ channel, except for Rand steps in the conductor, therefore the lemma is fulfilled. □

Lemma C.8.10 (Sync step completeness)**.**
If $\mathcal{E}\left[\!\!\left[\dfrac{\mathcal{D}_1}{\Gamma \vdash P_1 \rhd \Delta_1} \right]\!\!\right] = Q_1$,
$P_{c1} \in \mathit{PC}(\Delta_1 \circ \Delta^\circ)$ *and* $Q_1|P_{c1} \rightarrow^\star Q_2 \rightarrow Q_3'$ *where Lemma C.8.1 on* $Q_2 \rightarrow Q_3'$ *uses the Rand case in a conductor*
$\dfrac{\mathcal{D}_2}{}$
then $\exists\ \Gamma \vdash P_2 \rhd \Delta_2\ , P_{c2} \in \mathit{PC}(\Delta_2 \circ \Delta^\circ)$
such that $P_1 \rightarrow P_2$, $\Delta_1 \rightarrow^{0/1} \Delta_2$ *and* $\mathcal{E}[\![\mathcal{D}_2]\!]|P_{c2} \rightarrow^\star Q_3$ *and* $Q_3' \rightarrow^\star Q_3$.

Proof: If $Q_2 \rightarrow Q_3'$ uses the Rand case in a conductor then that conductor C must be generated from the type $\{l : G_l\}_{l \in L;L'}$. This means that

$$\begin{aligned} C \;=\; & \mathtt{out}_{\tilde{s}1} \rhd \{\mathtt{cases}_{L_1 \cup L'} : \ldots : \mathtt{out}_{\tilde{s}n} \rhd \{\mathtt{cases}_{L_n \cup L'} : \\ & \quad \mathtt{rand}\{\mathtt{in}_{\tilde{s}1} \lhd l; \ldots; \mathtt{in}_{\tilde{s}n} \lhd l; \mathcal{C}[\![G_l]\!]^\star_{n,\tilde{s}}\}_{l \in \bigcap_{i=1}^n L_i \cup L'}\}_{L_n \subseteq L} \ldots\}_{L_1 \subseteq L} \end{aligned}$$

Since we know that $Q_1|P_{c1} \rightarrow^\star Q_2$ and Q_2 steps using the Rand case in a conductor, we know that there must be processes in Q_1 sending on $\mathtt{out}_{\tilde{s}p}$ for all $p = 1..n$. They can only be created from Sync rules, since $Q_2 \rightarrow Q_3'$ is the first non-conduction step. This means that the processes

$$Q_{1p} = \mathtt{out}_{\tilde{s}p} \lhd \mathtt{cases}_{L''}; \mathtt{in}_{\tilde{s}p} \rhd \{l : P_{pl}\}_{l \in L_p} \text{ for } p = 1..n$$

can send to C, and this means that

$$Q_1|P_{c1} \equiv (\nu \tilde{n}')\mathsf{def}\ D'\ \mathsf{in}\ (C|Q_{11}|\ldots|Q_{1n}|\mathtt{in}_{\tilde{s}1} : \varnothing|\ldots|\mathtt{out}_{\tilde{s}n} : \varnothing|Q_1')$$
and
$$P_1 \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ ((\mathtt{sync}_{\tilde{s},n}\{l : P_{l1}\}_{l \in L1 \cup L'}|\ldots|\mathtt{sync}_{\tilde{s},n}\{l : P_{ln}\}_{l \in Ln \cup L'})|P_1'),$$

Now the extension of Theorem 5.22(1) from [61] in Appendix 4.4.2 gives us a derivation

$$\frac{\mathcal{D}_1'}{\Gamma \vdash (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ ((\mathtt{sync}_{\tilde{s},n}\{l : P_{l1}\}_{l \in L1 \cup L'}|\ldots|\mathtt{sync}_{\tilde{s},n}\{l : P_{ln}\}_{l \in Ln \cup L'})|P_1') \rhd \Delta_1}$$

such that $\mathcal{E}[\![\mathcal{D}_1]\!] \equiv \mathcal{E}[\![\mathcal{D}_1']\!]$, and we can see $\mathcal{D}_1'$ must contain a subderivation on the form

$$[\textsc{Par}]\quad \mathcal{D}_{11} = \left[\ \frac{[\textsc{Sync}]\quad \Gamma' \vdash \mathtt{sync}_{\tilde{s},n}\{l : P_{l1}\}_{l \in L1 \cup L'} \rhd \Delta_{11}, \tilde{s} : \{\{l : T_{l1}\}_{l \in L;L'}@(1,n)\}}{\Gamma' \vdash \mathtt{sync}_{\tilde{s},n}\{l : P_{l1}\}_{l \in L1 \cup L'} \rhd \Delta_{11}, \tilde{s} : \{\{l : T_{l1}\}_{l \in L;L'}@(1,n)\}}\ \right] \frac{[\textsc{Par}]\quad \vdots}{\vdots}$$

$$\Gamma' \vdash \mathtt{sync}_{\tilde{s},n}\{l : P_{l1}\}_{l \in L1 \cup L'}|\ldots|\mathtt{sync}_{\tilde{s},n}\{l : P_{ln}\}_{l \in Ln \cup L'} \rhd \Delta_1', \tilde{s} : \{\{l : T_{lp}\}_{l \in Lp \cup L'}@(p,n)\}_{p=1..n}$$

We get from Theorem 4.4.1 that $Q_1|P_{c1}$ is typable, and therefore it is linear. This means that we can propagate the sending and receiving on the $\mathtt{out}_{\tilde{s}p}$ channels and the Rand step forward to the beginning of the evaluation, giving us the evaluation

$$\begin{aligned}
&Q_1|P_{c1} \rightarrow^\star Q_2' \rightarrow Q_3'' \rightarrow^\star Q_3' \text{ where}\\
&Q_1|P_{c1} \equiv (\nu\tilde{n}')\mathsf{def}\ D' \ \mathsf{in}\ (C|Q_{11}|\dots|Q_{1n}|\mathtt{in}_{\tilde{s}1}:\varnothing|\dots|\mathtt{out}_{\tilde{s}n}:\varnothing|Q_1'),\\
&Q_2' \equiv (\nu\tilde{n}')\mathsf{def}\ D' \ \mathsf{in}\ (\mathtt{rand}\{\mathtt{in}_{\tilde{s}1} \lhd l;\dots;\mathtt{in}_{\tilde{s}n} \lhd l; \mathcal{C}[\![G_l]\!]^\star_{n,\tilde{s}}\}_{l\in\bigcap_{i=1}^n L_i \cup L'}\\
&\qquad\qquad |\mathtt{in}_{\tilde{s}1} \rhd \{l : P_{1l}\}_{l\in L_1}|\dots|\mathtt{in}_{\tilde{s}n} \rhd \{l : P_{nl}\}_{l\in L_n}\\
&\qquad\qquad |\mathtt{in}_{\tilde{s}1}:\varnothing|\dots|\mathtt{out}_{\tilde{s}n}:\varnothing|Q_1') \text{ and}\\
&Q_3'' \equiv (\nu\tilde{n}')\mathsf{def}\ D' \ \mathsf{in}\ (\mathtt{in}_{\tilde{s}1} \lhd l;\dots;\mathtt{in}_{\tilde{s}n} \lhd l; \mathcal{C}[\![G_l]\!]^\star_{n,\tilde{s}}\\
&\qquad\qquad |\mathtt{in}_{\tilde{s}1} \rhd \{l : P_{1l}\}_{l\in L_1}|\dots|\mathtt{in}_{\tilde{s}n} \rhd \{l : P_{nl}\}_{l\in L_n}\\
&\qquad\qquad |\mathtt{in}_{\tilde{s}1}:\varnothing|\dots|\mathtt{out}_{\tilde{s}n}:\varnothing|Q_1')
\end{aligned}$$

Now we can see that

$$Q_3'' \rightarrow^\star (\nu\tilde{n}')\mathsf{def} D' \mathsf{in}(\mathcal{C}[\![G_l]\!]^\star_{n,\tilde{s}}|P_{1l}|\dots|P_{nl}|\mathtt{in}_{\tilde{s}1}:\varnothing|\dots|\mathtt{out}_{\tilde{s}n}:\varnothing|Q_1') = Q_4$$

by sending and receiving on the $\mathtt{in}_{\tilde{s}p}$ channels. Lemma C.8.7 therefore gives us a Q_3 such that $Q_3' \rightarrow^\star Q_3$ and $Q_4 \rightarrow^\star Q_3$.

This means we only need to find $\dfrac{\mathcal{D}_2}{\Gamma \vdash P_2 \rhd \Delta_2}$ and $P_{c2} \in \mathtt{PC}(\Delta_2 \circ \Delta^\circ)$ such that $P_1 \rightarrow P_2$, $\Delta_1 \rightarrow^{0/1} \Delta_2$ and $\mathcal{E}[\![\mathcal{D}_2]\!]|P_{c2} \equiv Q_4$.
If we create the derivation

$$\dfrac{\text{[PAR]}\quad \mathcal{D}_{l1}\qquad\qquad \dfrac{\text{[PAR]}\quad \mathcal{D}_{l_{n-1}}\ \mathcal{D}_{ln}}{\cdots}}{\Gamma' \vdash P_{l1}|\dots|P_{ln} \rhd \Delta_1', \tilde{s} : \{T_{lp}@(p,n)\}_{p=1..n}}$$

and substitute this for the original subderivation in $\mathcal{D}_1'$ we get a derivation of $\dfrac{\mathcal{D}_2}{\Gamma \vdash P_2 \rhd \Delta_2}$ where $P_2 \equiv (\nu\tilde{n})\mathsf{def}\ D \ \mathsf{in}\ (P_{l1}|\dots|P_{ln}|P_1')$.
There are two possibilities for Δ_2. Either $\Delta_1 = \Delta_1', \tilde{s} : \{\{l : T_{lp}\}_{l\in L;L'}@(p,n)\}_{p=1..n}$ and $\Delta_2 = \Delta_1', \tilde{s} : \{T_{lp}@(p,n)\}_{p=1..n}$ which means that $\Delta_1 \rightarrow \Delta_2$ and we can select P_{c2} as P_{c1} where C has been replaced by $\mathcal{C}[\![G_l]\!]^\star_{\tilde{s},n}$. Otherwise $\tilde{s}$ is captured in P_1 and P_2 and in this case $\Delta_1 = \Delta_2$ and we can select $P_{c2} = P_{c1}$ but this means that the conductor generated by the capturing CRes rule will no longer be C but $\mathcal{C}[\![G_l]\!]^\star_{\tilde{s},n}$.
We can see that $P_1 \rightarrow P_2$ in a Sync step, and in both cases $\mathcal{E}[\![\mathcal{D}_2]\!]|P_{c2} \equiv Q_4$. □

Lemma C.8.11 (Single-step completeness)**.**
If $\mathcal{E}\left[\!\!\left[\dfrac{\mathcal{D}_1}{\Gamma \vdash P_1 \rhd \Delta_1} \right]\!\!\right] = Q_1$, $P_{c1} \in \mathtt{PC}(\Delta_1 \circ \Delta^\circ)$ *and* $Q_1|P_{c1} \rightarrow^\star Q_2 \rightarrow Q_3'$

$$\frac{\mathcal{D}_2}{\text{then } \exists\ \Gamma \vdash P_2 \rhd \Delta_2\ , P_{c2} \in \mathsf{PC}(\Delta_2 \circ \Delta^\circ)}$$
such that $P_1 \to P_2$, $\Delta_1 \to^{0/1} \Delta_2$ *and* $\mathcal{E}\,[\![\mathcal{D}_2]\!]\,|P_{c2} \rightarrow^\star Q_3$ *and* $Q_3' \rightarrow^\star Q_3$.

Proof: There are two cases.
If $Q_2 \rightarrow Q_3'$ uses the Rand case in a conductor then Lemma C.8.10 concludes the desired.
If $Q_2 \rightarrow Q_3'$ does not use the Rand case in a conductor, then Lemma C.8.9 gives a derivation of $Q_1|P_{c1} \rightarrow Q_2' \rightarrow^\star Q_3'$. Therefore Lemma C.8.8 gives us $\dfrac{\mathcal{D}_2}{\Gamma \vdash P_2 \rhd \Delta_2}$ and $P_{c2} \in \mathsf{PC}(\Delta_2 \circ \Delta^\circ)$ such that $P_1 \to P_2$, $\Delta_1 \to^{0/1} \Delta_2$ and $Q_2' \rightarrow^\star \mathcal{E}\,[\![\mathcal{D}_2]\!]\,|P_{c2}$. Now we have that $Q_2' \rightarrow^\star Q_3'$ and $Q_2' \rightarrow^\star \mathcal{E}\,[\![\mathcal{D}_2]\!]\,|P_{c2}$ and therefore we get from Lemma C.8.7 that there is a Q_3 such that $Q_3' \rightarrow^\star Q_3$ and $\mathcal{E}\,[\![\mathcal{D}_2]\!]\,|P_{c2} \rightarrow^\star Q_3$. □

Step 4: Sequencing steps

Proof C.8.12 (Theorem 4.4.4: Semantic completeness)**.**
If $\mathcal{E}\left[\!\!\left[\dfrac{\mathcal{D}_1}{\Gamma \vdash P_1 \rhd \varnothing} \right]\!\!\right] \rightarrow^\star Q'$
$$\frac{\mathcal{D}_2}{\text{then } \exists\ \Gamma \vdash P_2 \rhd \varnothing}$$
and Q *such that* $P_1 \rightarrow^\star P_2$ *and* $\mathcal{E}\,[\![\mathcal{D}_2]\!] \rightarrow^\star Q$ *and* $Q' \rightarrow^\star Q$.
Proof: *By induction on the number of non-conduction steps in* $\mathcal{E}\,[\![\mathcal{D}_1]\!] \rightarrow^\star Q'$.
If $P_1 \rightarrow^\star Q'$ *then the theorem is trivially fulfilled using* $\mathcal{D}_2 = \mathcal{D}_1$ *and* $Q = Q'$.
Otherwise we have that $\mathcal{E}\,[\![\mathcal{D}_1]\!] \rightarrow^\star Q_3' \rightarrow Q_2' \rightarrow^\star Q'$.
We obtain from the induction hypothesis that $\exists\ \dfrac{\mathcal{D}_3}{\Gamma \vdash P_3 \rhd \varnothing}$ *and* Q_3'' *such that* $P_1 \rightarrow^\star P_3$ *and* $\mathcal{E}\,[\![\mathcal{D}_3]\!] \rightarrow^\star Q_3''$ *and* $Q_3' \rightarrow^\star Q_3''$.
Now we have that $Q_3' \rightarrow Q_2'$ *and* $Q_3' \rightarrow^\star Q_3''$ *and therefore we get from Lemma C.8.6 that* $\exists Q_2''$ *such that* $Q_3'' \to Q_2''$ *and* $Q_2' \rightarrow^\star Q_2''$.
Now we have that $Q_2' \rightarrow^\star Q'$ *and* $Q_2' \rightarrow^\star Q_2''$ *and therefore we get from Lemma C.8.7 that* $\exists Q''$ *such that* $Q' \rightarrow^\star Q''$ *and* $Q_2' \rightarrow^\star Q''$.
We can now use Lemma C.8.11 on the found derivation of $\mathcal{E}\,[\![\mathcal{D}_3]\!] \rightarrow^\star Q_3 \rightarrow Q_2''$ *to find*
$$\frac{\mathcal{D}_2}{\Gamma \vdash P_2 \rhd \varnothing}$$
and Q_2''' *such that* $P_3 \to P_2$, $Q_2'' \rightarrow^\star Q_2'''$ *and* $\mathcal{E}\,[\![\mathcal{D}_2]\!] \rightarrow^\star Q_2'''$.
Finally we have that $Q_2'' \rightarrow^\star Q''$ *and* $Q_2'' \rightarrow^\star Q_2'''$ *and therefore Lemma C.8.7 gives us a* Q *such that* $Q'' \rightarrow^\star Q$ *and* $Q_2''' \rightarrow^\star Q$.
Therefore we now have that $P_1 \rightarrow^\star P_2$, $Q' \rightarrow^\star Q'' \rightarrow^\star Q$ *and* $\mathcal{E}\,[\![\mathcal{D}_2]\!] \rightarrow^\star Q_2''' \rightarrow^\star Q$. □
There are a lot of sub-steps in this proof, so to give an overview of the found evaluations,

we have included an illustration showing what lemmas are used to find the different parts of the evaluations in Figure 69.

C.9 ENCODABILITY CRITERIA

We now consider each criteria from Section 4.4.3 stemming from [51], proving that they are fulfilled by the presented erasure. This work culminates in Proof C.9.9 concluding that all the criteria are fulfilled, thus proving Theorem 4.4.6. Finally we prove that the relation we use for $\asymp_2$ is a weak barbed reduction congruence.

Lemma C.9.1 (Compositionality). *For every* k*-ary typing rule* R *in the typingsystem* $\Gamma \vdash P \triangleright \Delta$ *and every subset of names* N *there exists a* k*-ary context* $C^N_{op}(\__1, \ldots, \__k)$ *such that, for all* $\mathcal{D}_1, \ldots, \mathcal{D}_k$ *with* $\mathrm{FN}(\mathcal{D}_1, \ldots, \mathcal{D}_k) = N$*, it holds that* $\mathcal{E}\left[\!\!\left[\dfrac{[R] \quad \Gamma \vdash P \triangleright \Delta}{\Gamma \vdash P \triangleright \Delta} \right]\!\!\right] = C^N_R([\![\mathcal{D}_1]\!], \ldots, [\![\mathcal{D}_k]\!])$ *where* $\left[\!\!\left[\dfrac{\mathcal{D}'}{\Gamma' \vdash P' \triangleright \Delta'} \right]\!\!\right] = \mathcal{E}\,[\![\mathcal{D}']\!]$ *and* $[\![\Gamma' \vdash a : G]\!] = \mathcal{C}\,[\![G]\!]^\star_{\tilde{s},n}$.

Proof: This is clear from inspecting the right-hand sides of $\mathcal{E}\,[\![]\!]$.
It should be noted that $\mathcal{C}\,[\![]\!]^\star$ takes the arguments $\tilde{s}$ and n which are not in the derivation $\Gamma \vdash a : G$, but we know that $n = \mathsf{max}(\mathsf{pid}(G))$, and $|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G))$ so only the names in $\tilde{s}$ are not allowed to be used. Fortunately $\tilde{s}$ is only used for aesthetic reasons, to use the same channel names as the other processes. It could be left out, if the conductor used the names in_p and out_p in stead of $\mathsf{in}_{\tilde{s}p}$ and $\mathsf{out}_{\tilde{s}p}$ which does not cause interference since the conductor only uses one session. □

Lemma C.9.2 (Divergence reflection). *If* $\dfrac{\mathcal{D}_1}{\Gamma \vdash P_1 \triangleright \varnothing}$ *and* $\mathcal{E}\,[\![\mathcal{D}_1]\!] \to^\omega$ *then* $P_1 \to^\omega$.

Proof: This follows from Theorem 4.4.4, because non-conduction steps must occur infinitely often in $\mathcal{E}\,[\![\mathcal{D}_1]\!] \to^\omega$.

First we must consider, if an evaluation $\mathcal{E}\,[\![\mathcal{D}_1]\!] \to^\omega$ can contain a sub-evaluation $Q \to^\omega$. Since each recursive step (def-unfolding) in non-conductor processes is a non-conduction step, this would mean there is a sub-evaluation $Q' \to^\omega$ where only conductor processes step. Since conductor processes only use recursion when the body receives a message from a non-conductor process, this is not possible. Therefore non-conduction steps must occur *infinitely often* in $\mathcal{E}\,[\![\mathcal{D}_1]\!] \to^\omega$.

Now assume that $P \not\to^\omega$. In this case there is an upper limit N such that P can make at most N steps. Because the non-conduction steps occur *infinitely often* in $\mathcal{E}\,[\![\mathcal{D}_1]\!] \to^\omega$, we can select enough steps from $Q \to^\omega$ to include $N + 1$ non-conduction steps. Now Theorem 4.4.4 gives us an evaluation $P \to^\star P'$, and this

evaluation will have $N+1$ steps. This is a contradiction because P could make at most N steps. Therefore $P \to^{\omega}$. □

Definition C.9.3 (Success sensitiveness ($\Downarrow$)).
Processes are extended by the $\surd$ constant.

The typing system is extended by the rule $\dfrac{[\textsc{success}] \quad \Gamma \vdash \surd \triangleright \varnothing}{\Gamma \vdash \surd \triangleright \varnothing}$.

The erasure is extended by $\mathcal{E}\left[\!\!\left[\dfrac{[\textsc{success}] \quad \Gamma \vdash \surd \triangleright \varnothing}{\Gamma \vdash \surd \triangleright \varnothing} \right]\!\!\right] = \surd$.

We say that $P \downarrow$ *if* $P \equiv \surd \mid P'$.
We say that $P \Downarrow$ *if* $\exists Q$ *such that* $P \to^{\star} Q$ *and* $Q \downarrow$.

Lemma C.9.4 (Erasure Success-preservation). $\mathcal{E}\,[\![\mathcal{D} :: P]\!] \downarrow$ *if and only if* $P \downarrow$.

PROOF: This is proved by induction on the derivation $\mathcal{D}$, since the conductors does not introduce $\surd$. □

Lemma C.9.5 (Success soundness). *If* $\dfrac{\mathcal{D}}{\Gamma \vdash P \triangleright \Delta}$ *and* $P \Downarrow$ *then* $\mathcal{E}\,[\![\mathcal{D}]\!] \Downarrow$.

PROOF: Since $\dfrac{\mathcal{D}}{\Gamma \vdash P \triangleright \varnothing}$ and $P \to^{\star} P'$ where $P' \downarrow$ Corollary C.7.3 reveals a derivation $\dfrac{\mathcal{D}'}{\Gamma \vdash P' \triangleright \Delta'}$ such that $\mathcal{E}\,[\![\mathcal{D}]\!] \to^{\star} \mathcal{E}\,[\![\mathcal{D}']\!]$. Thus Lemma C.9.4 gives us that $\mathcal{E}\,[\![\mathcal{D}']\!] \downarrow$ and therefore this lemma is fulfilled. □

Lemma C.9.6 (Stepping success-preservation). *If* $P \downarrow$ *and* $P \to Q$ *then* $Q \downarrow$.

PROOF: $\downarrow$ is preserved by all equivalence and stepping rules. □

Lemma C.9.7 (Conduction stepping success-preservation). *If* $\mathcal{E}\,[\![\mathcal{D}]\!] = Q$ *and* $Q \rightarrow Q'$ *where* $Q' \downarrow$ *then* $Q \downarrow$.

PROOF: By induction on the call-tree of $\mathcal{E}\,[\![]\!]$ and $\mathcal{C}\,[\![]\!]$.
First we can see by inspection of all right-hand-sides of $\mathcal{E}\,[\![]\!]$ and $\mathcal{C}\,[\![]\!]$ that if $\mathtt{in}_{\tilde{s}p} : h$ is in Q then $h = \varnothing$.
Second we can see by inspection of all right-hand-sides of $\mathcal{E}\,[\![]\!]$ and $\mathcal{C}\,[\![]\!]$ that writing on any $\mathtt{in}$ channel is preceded by a non-conduction step.
Finally we can see by inspection of all right-hand-sides of $\mathcal{E}\,[\![]\!]$ and $\mathcal{C}\,[\![]\!]$ that either the result is equivalent to $\surd|P$ for some P, or all occurrences of $\surd$ are preceded by a non-conduction step or reading on an $\mathtt{in}$ channel. Therefore the lemma is fulfilled. □

Lemma C.9.8 (Success completeness). *If* $\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd \Delta}$ *and* $\mathcal{E}\,[\![\mathcal{D}]\!] \Downarrow$ *then* $P \Downarrow$.

PROOF: Since $\mathcal{E}\,[\![\mathcal{D}]\!] \to^\star Q'$ where $Q' \downarrow$ Theorem 4.4.4 reveals $\dfrac{\mathcal{D}'}{\Gamma \vdash P' \rhd \Delta'}$ and Q'' such that $P \to^\star P'$, $Q' \to^\star Q''$ and $\mathcal{E}\,[\![\mathcal{D}']\!] \to^\star Q''$. Lemma C.9.6 now proves that $Q'' \downarrow$ and therefore $\mathcal{E}\,[\![\mathcal{D}']\!] \downarrow$ because of Lemma C.9.7. Thus $P' \downarrow$ because of Lemma C.9.4 and therefore $P \Downarrow$. □

Proof C.9.9 (Theorem 4.4.6: Erasure fulfils criteria)**.**
The encodability criteria for the erasure mapping are fulfilled.

Compositionality: See Proof C.9.1 in the Appendix.

Name invariance: This criteria is fulfilled, because the only renaming we perform is to reserve the conductor channels, and [51] argues that the criteria allows this.

Operational correspondence: Completeness follows from Corollary C.7.3 using $Q' = \mathcal{E}\,[\![\mathcal{D}_2]\!]$*, and soundness follows from Theorem 4.4.4 using* $Q = Q_1$.

Divergence reflection: See Proof C.9.2 in the Appendix.

Success sensitiveness: See Lemma C.9.5 and Lemma C.9.8 in the Appendix. □

$\asymp_2$ *is a weak barbed reduction congruence*

Definition C.9.10 (Context and completion)**.**
$C ::= \cdot \mid C|P \mid (\nu n)C$

A completion of a context $C[P]$ *is a process defined by*
$\cdot[P_1] = P_1$
$(C|P)[P_1] = (C[P_1])|P$
$(\nu n)C[P_1] = (\nu n)(C[P_1])$ *if* $n \notin \mathrm{fn}(P_1)$.

Lemma C.9.11 ($\asymp_2$ is a Congruence relation)**.**
$P_1 \asymp_2 P_2 \Rightarrow C[P_1] \asymp_2 C[P_2]$.

PROOF: Structural induction on C.

Case: $C = \cdot$
In this case $C[P_1] = P_1$ and $C[P_2] = P_2$ and therefore we get from the assumption that $C[P_1] \asymp_2 C[P_2]$.

Case: $C = C'|P$
In this case the induction hypothesis gives us that $C'[P_1] \asymp_2 C'[P_2]$ which means that there is a Q such that $C'[P_1] \to^\star Q$ and $C'[P_2] \to^\star Q$. Now we can get evaluations of $C'[P_1]|P \to^\star Q|P$ and $C'[P_2]|P \to^\star Q|P$ by applying the rule Par to each step in both evaluations. Therefore we have that $C[P_1] \asymp_2 C[P_2]$.

Case: $C = (\nu n')C'$

In this case the induction hypothesis gives us that $C'[P_1] \asymp_2 C'[P_2]$ which means that there is a Q such that $C'[P_1] \rightarrow^\star Q$ and $C'[P_2] \rightarrow^\star Q$. Now we can get evaluations of $(\nu n)(C'[P_1]) \rightarrow^\star (\nu n)Q$ and $(\nu n)(C'[P_2]) \rightarrow^\star (\nu n)Q$ by applying the rule Scop to each step of both evaluations. Therefore we have that $C[P_1] \asymp_2 C[P_2]$. □

Definition C.9.12 (Commitment)**.**
$P \triangledown a$ *if* P *is ready to connect on channel* a*, that is*
$P \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (\overline{a}[2..p](\tilde{s}).P_1 | P_2)$ *or* $P \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (a[p](\tilde{s}).P_1 | P_2)$
where $a \notin \tilde{n}$.

$P \bigtriangledown a$ *if* $\exists P'.P \rightarrow^\star P'$ *and* $P' \triangledown a$.

Lemma C.9.13 ($\rightarrow$ preserves $\triangledown$)**.**
If $P_1 \rightarrow P_1'$ *and* $P_1 \triangledown a$ *then* $P_1' \triangledown a$.

Proof: There are three cases for $P_1 \rightarrow P_1'$

Case **Label**: In this case
$P_1 \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \lhd l; Q_1 | s_k : \tilde{h} | Q_2)$,
$P_1' \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1 | s_k : \tilde{h} \cdot l | Q_2)$
and s_k is a conductor channel.
If $P_1 \triangledown a$ then $Q_2 \triangledown a$ and $a \notin \tilde{n}$ and therefore $P_1' \triangledown a$.

Case **Branch**: In this case
$P_1 \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (s_k \rhd \{l : Q_1, \ldots\} | s_k : l \cdot \tilde{h} | Q_2)$,
$P_1' \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1 | s_k : \tilde{h} | Q_2)$
and s_k is a conductor channel.
If $P_1 \triangledown a$ then $Q_2 \triangledown a$ and $a \notin \tilde{n}$ and therefore $P_1' \triangledown a$.

Case **Def**: In this case
$P_1 \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (X\langle \tilde{e}\tilde{s} \rangle | Q_2)$,
$P_2 \equiv (\nu \tilde{n})\mathsf{def}\ D\ \mathsf{in}\ (Q_1[\tilde{v}/\tilde{x}] | Q_2)$ where $X(\tilde{x}\tilde{s}) = Q_1 \in D$ and $\tilde{e} \triangledown \tilde{v}$
and Q_1 is produced by $\mathcal{C}[\![]\!]^\star$.
If $P_1 \triangledown a$ then $Q_2 \triangledown a$ and $a \notin \tilde{n}$ and therefore $P_1' \triangledown a$. □

Lemma C.9.14 ($\rightarrow$ preserves $\bigtriangledown$)**.**
If P_1 *is linear,* $P_1 \rightarrow P_1'$ *and* $P_1 \bigtriangledown a$ *then* $P_1' \bigtriangledown a$.

Proof: If $P_1 \bigtriangledown a$ then there is a Q such that $P_1 \rightarrow^\star Q$ and $Q \triangledown a$. Since $P_1 \rightarrow P_1'$ Lemma C.8.7 gives us that there is a Q' such that $P_1' \rightarrow^\star Q'$ and $Q \rightarrow^\star Q'$. Now we can apply Lemma C.9.13 on each step of $Q \rightarrow^\star Q'$ to get that $Q' \triangledown a$, and therefore $P_1' \bigtriangledown a$. □

Lemma C.9.15 (Weak barbed bisimulation)**.**
If $P_1 \asymp_2 P_2$ *and* $P_1 \bigtriangledown a$ *then* $P_2 \bigtriangledown a$.

Proof: Since $P_1 \asymp_2 P_2$ there is a Q such that $P_1 \rightarrow^\star Q$ and $P_2 \rightarrow^\star Q$. Since $P_1 \bigtriangledown a$ we can use Lemma C.9.14 on each step of $P_1 \rightarrow^\star Q$ to get that $Q \bigtriangledown a$. This means there

is a Q' such that $Q \to^\star Q'$ and $Q' \triangledown s$. Now we have an evaluation $P_2 \to^\star Q \to^\star Q'$ which means that $P_2 \bigtriangledown a$. □

Lemma C.9.16 ($\to$ preserves $\asymp_2$)**.**
If $P_1 \asymp_2 P_2$ *and* $P_1 \to P_1'$ *then* $P_2 \to^\star P_2'$ *such that* $P_1' \asymp_2 P_2'$.

Proof: Assume $P_1 \asymp_2 P_2$ and $P_1' \to P_2'$. Since $P_1 \asymp_2 P_2$ there is a Q such that $P_1 \to^\star Q$ and $P_2 \to^\star Q$. Now Lemma C.8.6 gives us a Q' such that $P_1' \to^\star Q'$ and $Q \to^{0/1} Q'$. Therefore we can compose the steps $P_2 \to^\star Q \to^{0/1} Q' = P_2'$ to get an evaluation fulfilling the lemma. □

Corollary C.9.17 ($\to^\star$ preserves $\asymp_2$)**.**
If $P_1 \asymp_2 P_2$ *and* $P_1 \to^\star P_1'$ *then* $P_2 \to^\star P_2'$ *such that* $P_1' \asymp_2 P_2'$.

Proof: Follows by induction on the number of steps in $P_1 \to^\star P_1'$ using Lemma C.9.16 in each step. □

C.10 HEALTHCARE EXAMPLE

We now provide more details for the healthcare examples presented in the paper. Figure 70 shows how the synchronisation can be partially specified by the nurse process. Figure 72 include the global type of the healthcare example processes from Section 5.2 and the local types for each of the participants. Figure 73 shows the full global type representing the cooperation described by the *Process Matrix* from Section 4.5, and Figure 74 shows the local type for the doctor. Proof C.10.1 constructs the typing derivation for the healthcare example processes from Section 5.2, and thus proves Proposition 4.3.1.

Proof C.10.1 (Proposition 4.3.1: Example is typed)**.** $a : \langle G \rangle \vdash P_D \mid P_N \mid P_P \rhd \varnothing$.
First we type the patient-process P_P*. To do this, we start by typing each end-case of the process with the matching case of the local type.*

Notice that all the end-cases for P_P *are the same, thus*
$P_{PDD} = P_{PDN} = P_{PND} = P_{PNN} = s?(\mathit{schedule}); r?(\mathit{result}); 0$*, and the local types for each of the end-cases are also identical, thus*
$T_{PDD} = T_{PDN} = T_{PND} = T_{PNN} = 2?\langle \tilde{S}_{\mathit{schedule}} \rangle; 3?\langle \tilde{S}_{\mathit{result}} \rangle; \mathsf{end}$*. Therefore the typing of each end-case* $\mathcal{D}_{PDD} = \mathcal{D}_{PDN} = \mathcal{D}_{PND} = \mathcal{D}_{PNN}$ *is the same. The derivation is given below.*

$$\dfrac{\dfrac{\dfrac{}{[\textsc{Inact}]}}{[\textsc{Rcv}]\quad \alpha : \langle \mathsf{G} \rangle, \mathit{schedule} : \tilde{\mathsf{S}}_{\mathit{schedule}}, \mathit{result} : \tilde{\mathsf{S}}_{\mathit{result}} \vdash 0 \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : \mathsf{end}@(1,3)}}{[\textsc{Rcv}]\quad \alpha : \langle \mathsf{G} \rangle, \mathit{schedule} : \tilde{\mathsf{S}}_{\mathit{schedule}} \vdash \mathsf{r}?(\mathit{result}); 0 \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : 3?\langle \tilde{\mathsf{S}}_{\mathit{result}} \rangle; \mathsf{end}@(1,3)}}{\alpha : \langle \mathsf{G} \rangle \vdash \mathsf{s}?(\mathit{schedule}); \mathsf{r}?(\mathit{result}); 0 \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : 2?\langle \tilde{\mathsf{S}}_{\mathit{schedule}} \rangle; 3?\langle \tilde{\mathsf{S}}_{\mathit{result}} \rangle; \mathsf{end}@(1,3)}$$

Now we can collect the end cases using the `Sync` *rule. This gives us* $\mathcal{D}_{\mathsf{PD}'}$*:*

$$\dfrac{[\textsc{Sync}] \qquad \mathcal{D}_{\mathsf{PDD}} \quad \mathcal{D}_{\mathsf{PDN}}}{\alpha : \langle \mathsf{G} \rangle \vdash \mathit{sync}_{(\mathsf{d},\mathsf{s},\mathsf{r}),3}\{\mathit{CaseDD} : \mathsf{P}_{\mathsf{PDD}}, \mathit{CaseDN} : \mathsf{P}_{\mathsf{PDN}}\} \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : \{\mathit{CaseDD} : \mathsf{T}_{\mathsf{PDD}}, \mathit{CaseDN} : \mathsf{T}_{\mathsf{PDN}}\}@(1,3)}$$

and $\mathcal{D}_{\mathsf{PN}'}$*:*

$$\dfrac{[\textsc{Sync}] \qquad \mathcal{D}_{\mathsf{PND}} \quad \mathcal{D}_{\mathsf{PNN}}}{\alpha : \langle \mathsf{G} \rangle \vdash \mathit{sync}_{(\mathsf{d},\mathsf{s},\mathsf{r}),3}\{\mathit{CaseND} : \mathsf{P}_{\mathsf{PND}}, \mathit{CaseNN} : \mathsf{P}_{\mathsf{PNN}}\} \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : \{\mathit{CaseND} : \mathsf{T}_{\mathsf{PND}}, \mathit{CaseNN} : \mathsf{T}_{\mathsf{PNN}}\}@(1,3)}$$

Next we can use the rule `Send` *on* $\mathcal{D}_{\mathsf{PD}'}$ *and* $\mathcal{D}_{\mathsf{PN}'}$ *which allows us to type the* `CaseD` *and* `CaseN` *branches. This gives us* $\mathcal{D}_{\mathsf{PD}}$*:*

$$\dfrac{[\textsc{Send}] \qquad \alpha : \langle \mathsf{G} \rangle \vdash e_{\mathit{data}} : \mathsf{S}_{\mathit{data}} \quad \mathcal{D}_{\mathsf{PD}'}}{\alpha : \langle \mathsf{G} \rangle \vdash \mathsf{d}!\langle e_{\mathit{data}} \rangle; \mathsf{P}_{\mathsf{PD}'} \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : 1!\langle \mathsf{S}_{\mathit{data}} \rangle; \mathsf{T}_{\mathsf{PD}'}@(1,3)}$$

and $\mathcal{D}_{\mathsf{PN}}$*:*

$$\dfrac{[\textsc{Send}] \qquad \alpha : \langle \mathsf{G} \rangle \vdash e_{\mathit{data}} : \mathsf{S}_{\mathit{data}} \quad \mathcal{D}_{\mathsf{PN}'}}{\alpha : \langle \mathsf{G} \rangle \vdash \mathsf{d}!\langle e_{\mathit{data}} \rangle; \mathsf{P}_{\mathsf{PN}'} \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : 1!\langle \mathsf{S}_{\mathit{data}} \rangle; \mathsf{T}_{\mathsf{PN}'}@(1,3)}$$

Again we collect the branches using the `Sync` *rule, which gives us* $\mathcal{D}_{\mathsf{P}'}$*.*

$$\dfrac{[\textsc{Sync}] \qquad \mathcal{D}_{\mathsf{PD}} \quad \mathcal{D}_{\mathsf{PN}}}{\alpha : \langle \mathsf{G} \rangle \vdash \mathit{sync}_{(\mathsf{d},\mathsf{s},\mathsf{r}),3}\{\mathit{CaseD} : \mathsf{P}_{\mathsf{PD}}, \mathit{CaseN} : \mathsf{P}_{\mathsf{PN}}\} \rhd (\mathsf{d}, \mathsf{s}, \mathsf{r}) : (\mathsf{G} \upharpoonright 1)@(1,3)}$$

Now we can type P_{P} *using the* `Mcast` *rule as* $\alpha : \langle \mathsf{G} \rangle \vdash \mathsf{P}_{\mathsf{P}} \rhd \varnothing$*.*

$$\mathcal{D}_{\mathsf{P}} = \dfrac{[\textsc{Mcast}] \qquad \mathcal{D}_{\mathsf{P}'}}{\alpha : \langle \mathsf{G} \rangle \vdash \mathsf{P}_{\mathsf{P}} \rhd \varnothing}$$

The derivations for P_{D} *and* P_{N} *are found similarly except that the* `Macc` *rule is used in stead of* `Mcast`*. Finally the derivations for each process are collected using the* `Conc` *rule twice which gives us that* $\alpha : \langle \mathsf{G} \rangle \vdash \mathsf{P}_{\mathsf{N}} \mid \mathsf{P}_{\mathsf{D}} \mid \mathsf{P}_{\mathsf{P}} \rhd \varnothing$*.* □

C.11 FULL ABSTRACTION

For many translations between process calculi, *full abstraction* is the best way to capture how the semantics is preserved by the translation. Unfortunately the erasure we have presented produces new processes to help the existing processes to

perform the synchronisation, and therefore it is important that the new processes actually behaves the way they have been defined. The problem is that *full abstraction* considers any context for the processes, and this allows processes to join the sessions as the new conductor process without behaving as the defined conductors.

For this reason *full abstraction* is not fulfilled for the presented erasure, and Observation C.11.1 gives an explicit example of this.

Observation C.11.1 (No full abstraction)**.**
Consider the processes

$$\begin{array}{rcl} P_1 & = & \overline{a}[2..3](\tilde{s}).\mathit{sync}_{\tilde{s},3}\{\underline{l} : b[2](\tilde{t}).0\} \\ & | & a[2](\tilde{s}).\mathit{sync}_{\tilde{s},3}\{\underline{l} : 0\} \\ & | & a[3](\tilde{s}).\mathsf{def}\ X(\tilde{s}) = X(\tilde{s})\ \mathsf{in}\ X(\tilde{s}) \end{array} \qquad \begin{array}{rcl} P_2 & = & \overline{a}[2..3](\tilde{s}).\mathit{sync}_{\tilde{s},3}\{l : 0\} \\ & | & a[2](\tilde{s}).\mathit{sync}_{\tilde{s},3}\{l : b[2](\tilde{t}).0\} \\ & | & a[3](\tilde{s}).\mathsf{def}\ X(\tilde{s}) = X(\tilde{s})\ \mathsf{in}\ X(\tilde{s}) \end{array}$$

It is fairly intuitive that $P_1 \approx P_2$*, because any context either allows the processes to synchronise (by providing a non-looping 3rd participant) in which case both processes can connect on channel* b*, or does not allow the processes to synchronise in which case none of the processes can connect on channel* b*.*

By using the erasure on the default typing derivations we get

$$\begin{array}{rcl} Q_1 & = & \overline{a}[2..4](\tilde{s}\tilde{c}).\mathit{out}_1 \lhd \mathit{cases}_{\{l\}}; \mathit{in}_1 \rhd \{l : b[2](\tilde{t}\tilde{c}).0\} \\ & | & a[2](\tilde{s}\tilde{c}).\mathit{out}_2 \lhd \mathit{cases}_{\{l\}}; \mathit{in}_2 \rhd \{l : 0\} \\ & | & a[3](\tilde{s}\tilde{c}).\mathsf{def}\ X(\tilde{s}\tilde{c}) = X(\tilde{s}\tilde{c})\ \mathsf{in}\ X(\tilde{s}\tilde{c}) \\ & | & a[4](\tilde{s}\tilde{c}).\mathit{out}_1 \rhd \{\mathit{cases}_{\{l\}} : \mathit{out}_2 \rhd \{\mathit{cases}_{\{l\}} : \mathit{out}_3 \rhd \{ \\ & & \mathit{cases}_{\{l\}} : \mathit{rand}\{\mathit{in}_1 \lhd l; \mathit{in}_2 \lhd l; \mathit{in}_3 \lhd l; 0\}\}\}\} \end{array}$$

$$\begin{array}{rcl} Q_2 & = & \overline{a}[2..4](\tilde{s}\tilde{c}).\mathit{out}_1 \lhd \mathit{cases}_{\{l\}}; \mathit{in}_1 \rhd \{l : 0\} \\ & | & a[2](\tilde{s}\tilde{c}).\mathit{out}_2 \lhd \mathit{cases}_{\{l\}}; \mathit{in}_2 \rhd \{l : b[2](\tilde{t}\tilde{c}).0\} \\ & | & a[3](\tilde{s}\tilde{c}).\mathsf{def}\ X(\tilde{s}\tilde{c}) = X(\tilde{s}\tilde{c})\ \mathsf{in}\ X(\tilde{s}\tilde{c}) \\ & | & a[4](\tilde{s}\tilde{c}).\mathit{out}_1 \rhd \{\mathit{cases}_{\{l\}} : \mathit{out}_2 \rhd \{\mathit{cases}_{\{l\}} : \mathit{out}_3 \rhd \{ \\ & & \mathit{cases}_{\{l\}} : \mathit{rand}\{\mathit{in}_1 \lhd l; \mathit{in}_2 \lhd l; \mathit{in}_3 \lhd l; 0\}\}\}\} \end{array}$$

But when using the context

$$C = \cdot \mid a[4](\tilde{s}\tilde{c}).\mathit{out}_1 \rhd \{\mathit{cases}_{\{l\}} : \mathit{in}_1 \lhd l; 0\}$$

We can see that it is possible for $C[Q_1]$ *but not for* $C[Q_2]$ *to connect on* b*, and therefore* $Q_1 \not\approx Q_2$*. Therefore the erasure does not fulfil* full abstraction*.* □

C.12 IMPLEMENTATION

We include APIMS code, which implements the processes from the healthcare example in Section 4.5 in C.12.1.

C.12.1 *Processes for example workflow*

```
(nu a: { ^Pdata: 1=>2:2<String>;1=>3:3<String>;rec $stateD.
           { ^Pdata: 1=>2:2<String>;1=>3:3<String>;$stateD,
             ^Dschedule: 2=>1:1<String>;2=>3:3<String>;rec $stateDS.
               { ^Pdata: 1=>2:2<String>;1=>3:3<String>;$stateD,
                 ^Dschedule: 2=>1:1<String>;2=>3:3<String>;$stateDS,
                 ^Nschedule: 3=>1:1<String>;3=>2:2<String>;$stateDS,
                 ^Dresult: 2=>1:1<String>;Gend
               },
             ^Nschedule: 3=>1:1<String>;3=>2:2<String>;rec $stateDS.
               { ^Pdata: 1=>2:2<String>;1=>3:3<String>;$stateD,
                 ^Dschedule: 2=>1:1<String>;2=>3:3<String>;$stateDS,
                 ^Nschedule: 3=>1:1<String>;3=>2:2<String>;$stateDS,
                 ^Dresult: 2=>1:1<String>;Gend
               }
           }
       })
( // Patient
  link(3,a,s,1);
  guisync(3,s,1)
  { ^Pdata(symptoms: String):
      s[2]<<symptoms;
      s[3]<<symptoms;
      def StateD(s: rec %stateD.
                    { ^Dschedule: 1>><String>;rec %stateDS.
                        { ^Dresult: 1>><String>;Lend,
                          ^Dschedule: 1>><String>;%stateDS,
                          ^Nschedule: 1>><String>;%stateDS,
                          ^Pdata: 2<<<String>;3<<<String>;%stateD
                        },
                      ^Nschedule: 1>><String>;rec %stateDS.
                        { ^Dresult: 1>><String>;Lend,
                          ^Dschedule: 1>><String>;%stateDS,
                          ^Nschedule: 1>><String>;%stateDS,
                          ^Pdata: 2<<<String>;3<<<String>;%stateD
                        },
                      ^Pdata: 2<<<String>;3<<<String>;%stateD
                    }@(1 of 3)) =
        def StateDS(s: rec %stateDS.
                       { ^Dresult: 1>><String>;Lend,
                         ^Dschedule: 1>><String>;%stateDS,
                         ^Nschedule: 1>><String>;%stateDS,
                         ^Pdata: 2<<<String>;3<<<String>;rec %stateD.
                           { ^Dschedule: 1>><String>;rec %stateDS.
                               { ^Dresult: 1>><String>;Lend,
                                 ^Dschedule: 1>><String>;%stateDS,
```

```
                                ^Nschedule: 1>><String>;%stateDS,
                                ^Pdata: 2<<<String>;3<<<String>;%stateD
                              },
                            ^Nschedule: 1>><String>;rec %stateDS.
                              { ^Dresult: 1>><String>;Lend,
                                ^Dschedule: 1>><String>;%stateDS,
                                ^Nschedule: 1>><String>;%stateDS,
                                ^Pdata: 2<<<String>;3<<<String>;%stateD
                              },
                            ^Pdata: 2<<<String>;3<<<String>;%stateD
                          }
                      }@(1 of 3)) =
          guisync(3,s,1)
          { ^Pdata(symptoms: String):
              s[2]<<symptoms;
              s[3]<<symptoms;
              StateD(s),
            ^Nschedule():
              s[1]>>schedule;
              StateDS(s),
            ^Dschedule():
              s[1]>>schedule;
              StateDS(s),
            ^Dresult():
              s[1]>>result;
              end
          }
        in guisync(3,s,1)
           { ^Pdata(symptoms: String):
               s[2]<<symptoms;
               s[3]<<symptoms;
               StateD(s),
             ^Dschedule():
               s[1]>>schedule;
               StateDS(s),
             ^Nschedule():
               s[1]>>schedule;
               StateDS(s)
           }
      in StateD(s)
  }
| // Doctor
  link(3,a,s,2);
  guisync(3,s,2)
  { ^Pdata():
      s[2]>>symptoms;
      def StateD(s: rec %stateD.
                    { ^Dschedule: 1<<<String>;3<<<String>;rec %stateDS.
                        { ^Dresult: 1<<<String>;Lend,
                          ^Dschedule: 1<<<String>;3<<<String>;%stateDS,
                          ^Nschedule: 2>><String>;%stateDS,
                          ^Pdata: 2>><String>;%stateD
                        },
                      ^Nschedule: 2>><String>;rec %stateDS.
                        { ^Dresult: 1<<<String>;Lend,
                          ^Dschedule: 1<<<String>;3<<<String>;%stateDS,
                          ^Nschedule: 2>><String>;%stateDS,
                          ^Pdata: 2>><String>;%stateD
                        },
```

```
                    ^Pdata: 2>><String>;%stateD
                  }@(2 of 3)) =
        def StateDS(s: rec %stateDS.
                      { ^Dresult: 1<<<String>;Lend,
                        ^Dschedule: 1<<<String>;3<<<String>;%stateDS,
                        ^Nschedule: 2>><String>;%stateDS,
                        ^Pdata: 2>><String>;rec %stateD.
                          { ^Dschedule: 1<<<String>;3<<<String>;rec %stateDS.
                              { ^Dresult: 1<<<String>;Lend,
                                ^Dschedule: 1<<<String>;3<<<String>;%stateDS,
                                ^Nschedule: 2>><String>;%stateDS,
                                ^Pdata: 2>><String>;%stateD
                              },
                            ^Nschedule: 2>><String>;rec %stateDS.
                              { ^Dresult: 1<<<String>;Lend,
                                ^Dschedule: 1<<<String>;3<<<String>;%stateDS,
                                ^Nschedule: 2>><String>;%stateDS,
                                ^Pdata: 2>><String>;%stateD
                              },
                            ^Pdata: 2>><String>;%stateD
                          }
                      }@(2 of 3)) =
          guisync(3,s,2)
          { ^Pdata():
              s[2]>>symptoms;
              StateD(s),
            ^Nschedule():
              s[2]>>schedule;
              StateDS(s),
            ^Dschedule(schedule: String):
              s[1]<<schedule;
              s[3]<<schedule;
              StateDS(s),
            ^Dresult(result: String):
              s[1]<<result;
              end
          }
        in guisync(3,s,2)
           { ^Pdata():
               s[2]>>symptoms;
               StateD(s),
             ^Dschedule(schedule: String):
               s[1]<<schedule;
               s[3]<<schedule;
               StateDS(s),
             ^Nschedule():
               s[2]>>schedule;
               StateDS(s)
           }
      in StateD(s)
  }
| // Nurse
  link(3,a,s,3);
  guisync(3,s,3)
  { ^Pdata():
      s[3]>>symptoms;
      def StateD(s: rec %stateD.
                   { ^Dschedule: 3>><String>;rec %stateDS.
                       { ^Dresult: Lend,
```

```
                    ^Dschedule: 3>><String>;%stateDS,
                    ^Nschedule: 1<<<String>;2<<<String>;%stateDS,
                    ^Pdata: 3>><String>;%stateD
                  },
                ^Nschedule: 1<<<String>;2<<<String>;rec %stateDS.
                  { ^Dresult: Lend,
                    ^Dschedule: 3>><String>;%stateDS,
                    ^Nschedule: 1<<<String>;2<<<String>;%stateDS,
                    ^Pdata: 3>><String>;%stateD
                  },
                ^Pdata: 3>><String>;%stateD
              }@(3 of 3)) =
  def StateDS(s: rec %stateDS.
                { ^Dresult: Lend,
                  ^Dschedule: 3>><String>;%stateDS,
                  ^Nschedule: 1<<<String>;2<<<String>;%stateDS,
                  ^Pdata: 3>><String>;rec %stateD.
                    { ^Dschedule: 3>><String>;rec %stateDS.
                        { ^Dresult: Lend,
                          ^Dschedule: 3>><String>;%stateDS,
                          ^Nschedule: 1<<<String>;2<<<String>;%stateDS,
                          ^Pdata: 3>><String>;%stateD
                        },
                      ^Nschedule: 1<<<String>;2<<<String>;rec %stateDS.
                        { ^Dresult: Lend,
                          ^Dschedule: 3>><String>;%stateDS,
                          ^Nschedule: 1<<<String>;2<<<String>;%stateDS,
                          ^Pdata: 3>><String>;%stateD
                        },
                      ^Pdata: 3>><String>;%stateD
                    }
                }@(3 of 3)) =
    guisync(3,s,3)
    { ^Pdata():
        s[3]>>symptoms;
        StateD(s),
      ^Nschedule(schedule: String):
        s[1]<<schedule;
        s[2]<<schedule;
        StateDS(s),
      ^Dschedule():
        s[3]>>schedule;
        StateDS(s),
      ^Dresult():
        end
    }
  in guisync(3,s,3)
     { ^Pdata():
         s[3]>>symptoms;
         StateD(s),
       ^Dschedule():
         s[3]>>schedule;
         StateDS(s),
       ^Nschedule(schedule: String):
         s[1]<<schedule;
         s[2]<<schedule;
         StateDS(s)
     }
in StateD(s)
```

```
  }
)
```

Figure 63 The typing rules ($\Gamma \vdash P \triangleright \Delta$)

$$\frac{[\textsc{Rand}] \quad \forall i \in I. \Gamma \vdash P_i \triangleright \Delta}{\Gamma \vdash \mathsf{rand}\{P_i\}_{i\in I} \triangleright \Delta} \qquad \frac{[\textsc{Name}]}{\Gamma, a : S \vdash a : S} \qquad \frac{[\textsc{Subs}] \quad \Gamma \vdash P \triangleright \Delta \quad \Delta \leqslant \Delta'}{\Gamma \vdash P \triangleright \Delta'}$$

$$\frac{[\textsc{Sync}] \quad \forall l \in L'' : \Gamma \vdash P_l \triangleright \Delta, \tilde{s} : T_l@(p, n) \quad L'' \subseteq L \cup L' \quad L' \subseteq L''}{\Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{l : P_l\}_{l\in L''} \triangleright \Delta, \tilde{s} : \{l : T_l\}_{l\in L;L'}@(p, n)}$$

$$\frac{[\textsc{Mcast}] \quad \Gamma \vdash a : \langle G\rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G\upharpoonright 1)@(1, n) \quad |\tilde{s}| = \max(\mathsf{sid}(G)) \quad n = \max(\mathsf{pid}(G))}{\Gamma \vdash \overline{a}[2..n](\tilde{s}).P \triangleright \Delta}$$

$$\frac{[\textsc{Macc}] \quad \Gamma \vdash a : \langle G\rangle \quad \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G\upharpoonright p)@(p, n) \quad |\tilde{s}| = \max(\mathsf{sid}(G)) \quad n = \max(\mathsf{pid}(G))}{\Gamma \vdash a[p](\tilde{s}).P \triangleright \Delta}$$

$$\frac{[\textsc{Send}] \quad \forall j. \Gamma \vdash e_j : S_j \quad \Gamma \vdash P \triangleright \Delta, s : T@(p, n)}{\Gamma \vdash s_k!\langle \tilde{e}\rangle; P \triangleright \Delta, \tilde{s} : k!\langle \tilde{S}\rangle; T@(p, n)}$$

$$\frac{[\textsc{Rcv}] \quad \Gamma, \tilde{x} : \tilde{S} \vdash P \triangleright \Delta, s : T@(p, n)}{\Gamma \vdash s_k?(\tilde{x}); P \triangleright \Delta, \tilde{s} : k?\langle \tilde{S}\rangle; T@(p, n)} \qquad \frac{[\textsc{If}] \quad \Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta}$$

$$\frac{[\textsc{Deleg}] \qquad \Gamma \vdash P \triangleright \Delta, \tilde{s} : T@(p, n)}{\Gamma \vdash s_k!\langle\!\langle \tilde{t}\rangle\!\rangle; P \triangleright \Delta, \tilde{s} : k!\langle T'@(p', |\tilde{t}|, n')\rangle; T@(p, n), \tilde{t} : T'@(p', n')}$$

$$\frac{[\textsc{Srec}] \qquad \Gamma \vdash P \triangleright \Delta, \tilde{s} : T@(p, n), \tilde{t} : T'@(p', n')}{\Gamma \vdash s_k?((\tilde{t})); P \triangleright \Delta, \tilde{s} : k?\langle T'@(p', |\tilde{t}|, n')\rangle; T@(p, n)}$$

$$\frac{[\textsc{Sel}] \qquad \Gamma \vdash P \triangleright \Delta, \tilde{s} : T_h@(p, n) \quad h \in L}{\Gamma \vdash s_k \lhd h; P \triangleright \Delta, \tilde{s} : k \oplus \{l : T_l\}_{l\in L}@(p, n)}$$

$$\frac{[\textsc{Branch}] \quad \forall l \in L : \quad \Gamma \vdash P_l \triangleright \Delta, \tilde{s} : T_l@(p, n)}{\Gamma \vdash s_k \rhd \{l : P_l\}_{l\in L} \triangleright \Delta, \tilde{s} : k?\{l : T_l\}_{l\in L}@(p, n)}$$

$$\frac{[\textsc{Conc}] \quad \Gamma \vdash P|Q \triangleright \Delta \circ \Delta'}{\Gamma \vdash P|Q \triangleright \Delta \circ \Delta'}\ (\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \varnothing) \qquad \frac{[\textsc{Inact}] \quad \Delta \quad \text{end only}}{\Gamma \vdash 0 \triangleright \Delta}$$

$$\frac{[\textsc{Nres}] \quad \Gamma, a : \langle G\rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta}$$

$$\frac{[\textsc{Var}] \qquad \Gamma \vdash \tilde{e} : \tilde{S} \quad \Delta \quad \text{end only}}{\Gamma, X : \tilde{S}\tilde{T} \vdash X\langle \tilde{e}\tilde{s}_1 ... \tilde{s}_{|\tilde{T}|}\rangle \triangleright \Delta, \tilde{s}_1 : T_1@(p_1, n_1), ..., \tilde{s}_n : T_{|\tilde{T}|}@(p_{|\tilde{T}|}, n_{|\tilde{T}|})}$$

$$\frac{[\textsc{Def}] \quad \Gamma, X : \tilde{S}\tilde{T}, \tilde{x} : \tilde{S} \vdash P \triangleright \tilde{s}_1 : T_1@(p_1, n_1), ..., \tilde{s}_{|\tilde{T}|} : T_{|\tilde{T}|}@(p_{|\tilde{T}|}, n_{|\tilde{T}|}) \quad \Gamma, X : \tilde{S}\tilde{T} \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{def}\ X(\tilde{x}\tilde{s}_1, ..., \tilde{s}_{|\tilde{T}|}) = P\ \mathsf{in}\ Q \triangleright \Delta}$$

Figure 64 Projection from global to local types ($\upharpoonright$)

$$(p_0 \to p_1 : k\langle U\rangle.G')\upharpoonright p = \begin{cases} m!\langle U\rangle;(G'\upharpoonright p) & \text{if } p = p_0 \text{ and } p \neq p_1 \\ m?\langle U\rangle;(G'\upharpoonright p) & \text{if } p = p_1 \text{ and } p \neq p_0 \\ G'\upharpoonright p & \text{if } p \neq p_0 \text{ and } p \neq p_1 \end{cases}$$

$$(p_0 \to p_1 : k\{l_j : G_j\}_{j\in J})\upharpoonright p = \begin{cases} k \oplus \{l_j : (G_j\upharpoonright p)\}_{j\in J} & \text{if } p = p_0 \neq p_1 \\ k \,\&\, \{l_j : (G_j\upharpoonright p)\}_{j\in J} & \text{if } p = p_1 \neq p_0 \\ \max_{\leqslant_{sub}}\{T' \mid \forall j \in J.T' \leqslant_{sub} (G_j\upharpoonright p)\} & \text{if } p_1 \neq p \neq p_2 \end{cases}$$

$$(\{l : G_l\}_{l\in L;L'})\upharpoonright p = \{l : (G_l\upharpoonright p)\}_{l\in L;L'} \quad (\mu t.G)\upharpoonright p = \mu t.(G\upharpoonright p) \quad t\upharpoonright p = t \quad \mathsf{end}\upharpoonright p = \mathsf{end}$$

Figure 65 Subtyping for local types

The subtyping for local types from [61, § 5], denoted $T \leqslant_{sub} T'$ is the *greatest fixed point* of the function S that maps each binary relation R on local types as regular trees to $S(R)$ given as

end $S(R)$ end
If TRT' then $k!\langle U\rangle;T$ $S(R)$ $k!\langle U\rangle;T'$ and $k?\langle U\rangle;T$ $S(R)$ $k?\langle U\rangle;T$
If $T_i R T'_i$, for each $i \in I \subseteq J$ then $k \oplus \{l_i : T_i\}_{i\in I}$ $S(R)$ $k \oplus \{l_j : T'_j\}_{j\in J}$
and $k\&\{l_j : T_j\}_{j\in J}$ $S(R)$ $k\&\{l_i : T'_i\}_{i\in I}$ and $\{l_i : T_i\}_{i\in I}$ $S(R)$ $\{l_i : T'_i\}_{i\in I}$

Figure 66 Conductor process generation from a global type (Step 2)

$$\begin{aligned}
\mathcal{C}[\![G]\!]_{\tilde{s},n,a} &= a[n+1](\tilde{s}, \mathtt{in}_{\tilde{s},1}, \mathtt{out}_{\tilde{s},1}, \ldots, \mathtt{in}_{\tilde{s},n}, \mathtt{out}_{\tilde{s},n}).\mathcal{C}[\![G]\!]^{\star}_{\tilde{s},n} \\
\mathcal{C}[\![p_1 \to p_2 : k\langle U\rangle.G']\!]^{\star}_{\tilde{s},n} &= \mathcal{C}[\![G']\!]^{\star}_{\tilde{s},n} \\
\mathcal{C}[\![p_1 \to p_2 : k\{l : G_l\}_{l\in L}]\!]^{\star}_{\tilde{s},n} &= \mathtt{out}_{\tilde{s},p_2} \rhd \{l : \mathcal{C}[\![G_l]\!]^{\star}_{\tilde{s},n}\}_{l\in L} \\
\mathcal{C}[\![\{l : G_l\}_{l\in L;L'}]\!]^{\star}_{\tilde{s},n} &= \mathtt{out}_{\tilde{s}1} \rhd \{\mathsf{cases}_{L_1\cup L'} : \ldots : \mathtt{out}_{\tilde{s}n} \rhd \{\mathsf{cases}_{L_n\cup L'} : \\
&\quad \mathsf{rand}\{\mathtt{in}_{\tilde{s}1} \lhd l; \ldots; \mathtt{in}_{\tilde{s}n} \lhd l; \mathcal{C}[\![G_l]\!]^{\star}_{n,\tilde{s}}\}_{l\in\bigcap_{i=1}^{n} L_i\cup L'} \\
&\quad \}_{L_n\subseteq L} \ldots\}_{L_1\subseteq L} \\
\mathcal{C}[\![\mu t.G']\!]^{\star}_{\tilde{s},n} &= \textbf{if } \mathcal{C}[\![G']\!]^{\star}_{\tilde{s},n} \neq X_t\langle\tilde{s}, \mathtt{in}_{\tilde{s}1}, \mathtt{out}_{\tilde{s}1}, \ldots, \mathtt{in}_{\tilde{s}n}, \mathtt{out}_{\tilde{s}n}\rangle \\
&\quad \textbf{then } \mathsf{def}\ X_t(\tilde{s}, \mathtt{in}_{\tilde{s}1}, \mathtt{out}_{\tilde{s}1}, \ldots, \mathtt{in}_{\tilde{s}n}, \mathtt{out}_{\tilde{s}n}) = \\
&\quad\quad \mathcal{C}[\![G']\!]^{\star}_{\tilde{s},n} \ \mathsf{in}\ X_t\langle\tilde{s}, \mathtt{in}_{\tilde{s}1}, \mathtt{out}_{\tilde{s}1}, \ldots, \mathtt{in}_{\tilde{s}n}, \mathtt{out}_{\tilde{s}n}\rangle \\
&\quad \textbf{else } 0 \\
\mathcal{C}[\![t]\!]^{\star}_{\tilde{s},n} &= X_t\langle\tilde{s}, \mathtt{in}_{\tilde{s}1}, \mathtt{out}_{\tilde{s}1}, \ldots, \mathtt{in}_{\tilde{s}n}, \mathtt{out}_{\tilde{s}n}\rangle \\
\mathcal{C}[\![\mathsf{end}]\!]^{\star}_{\tilde{s},n} &= 0
\end{aligned}$$

Figure 67 Type erasure mappings (Step 3)

Global Type Translation

$$
\begin{aligned}
[\![G]\!] &= [\![G]\!]^{\star}_{\mathsf{max}(\mathsf{pid}(G)),\mathsf{max}(\mathsf{sid}(G))} \\
\\
[\![p_0 \to p_1 : k\langle U\rangle.G']\!]^{\star}_{n,m} &= p_0 \to p_1 : k\langle [\![U]\!]\rangle.\, [\![G']\!]^{\star}_{n,m} \\
[\![p_0 \to p_1 : k\{l : G_l\}_{l\in L}]\!]^{\star}_{n,m} &= p_0 \to p_1 : k\{l_j : \\
&\quad p_0 \to (n{+}1) : (m + 2\cdot p_0)\{l : [\![G_l]\!]^{\star}_{n,m}\}\}_{l\in L} \\
[\![\{l : G_l\}_{l\in L;L'}]\!]^{\star}_{n,m} &= 1 \to n{+}1 : (m+2)\{cases_{L_1\cup L'} : \\
&\quad 2 \to n{+}1 : (m+4)\{cases_{L_2\cup L'} : \ldots \\
&\quad n \to n{+}1 : (m + 2\cdot n)\{cases_{L_n\cup L'} : \\
&\quad n{+}1 \to 1 : (m+1)\{l : \\
&\quad n{+}1 \to 2 : (m+3)\{l : \ldots \\
&\quad n{+}1 \to n : (m + 2\cdot n - 1)\{l : [\![G_l]\!]^{\star}_{n,m}\}\ldots\} \\
&\quad \}_{l\in\bigcap_{i=0}^{n} L_i\cup L'}\}_{L_n\subseteq L}\ldots\}_{L_1\subseteq L} \\
[\![\mu t.G']\!]^{\star}_{n,m} &= \mu t.\, [\![G']\!]^{\star}_{n,m} \\
[\![t]\!]^{\star}_{n,m} &= t \\
[\![\mathsf{end}]\!]^{\star}_{n,m} &= \mathsf{end}
\end{aligned}
$$

Local Type Translation

$$
\begin{aligned}
[\![k!\langle U\rangle;T']\!]_{n,m,p} &= k!\langle [\![U]\!]\rangle; [\![T']\!]_{n,m,p} \\
[\![k?\langle U\rangle;T']\!]_{n,m,p} &= k?\langle [\![U]\!]\rangle; [\![T']\!]_{n,m,p} \\
[\![k \oplus \{l : T_l\}_{l\in L}]\!]_{n,m,p} &= k \oplus \{l : (m + 2\cdot p) \oplus \{l : [\![T_l]\!]_{n,m,p}\}\}_{l\in L} \\
[\![k\&\{l : T_l\}_{l\in L}]\!]_{n,m,p} &= k\&\{l : [\![T_l]\!]_{n,m,p}\}_{l\in L} \\
[\![\mu t.T']\!]_{n,m,p} &= \mu t.\, [\![T']\!]_{n,m,p} \\
[\![t]\!]_{n,m,p} &= t \\
[\![\mathsf{end}]\!]_{n,m,p} &= \mathsf{end} \\
[\![\{l : T_l\}_{l\in L;L'}]\!]_{n,m,p} &= (m + 2\cdot p) \oplus \{cases_{L''\cup L'} : \\
&\quad (m + 2\cdot p - 1)\&\{l : [\![T_l]\!]_{n,m,p}\}_{l\in L''\cup L'}\}_{L''\subseteq L}
\end{aligned}
$$

Message Type Translation

$$
\begin{aligned}
[\![\tilde{S}]\!] &= \tilde{S} \\
[\![T@(p,m,n)]\!] &= [\![T]\!]_{n,m,p}@(p, m+1)
\end{aligned}
$$

Global Environment Translation

$$
\begin{aligned}
[\![\varnothing]\!] &= \varnothing \\
[\![\Gamma, u : \langle G\rangle]\!] &= [\![\Gamma]\!], u : \langle [\![G]\!]\rangle \\
[\![\Gamma, X : \tilde{S}\Delta]\!] &= [\![\Gamma]\!], X : [\![\tilde{S}]\!]\, [\![\Delta]\!]
\end{aligned}
$$

Local Environment Translation

$$
\begin{aligned}
[\![\varnothing]\!] &= \varnothing \\
[\![\Delta, \tilde{s} : T@(p,n)]\!] &= [\![\Delta]\!], (\tilde{s}, \mathsf{in}_{\tilde{s}1}, \ldots, \mathsf{out}_{\tilde{s}n}) : [\![T]\!]_{|\tilde{s}|,p}@(p, n+1)
\end{aligned}
$$

Figure 68 Erasure extension to runtime processes

$$\mathcal{E}\left[\!\!\left[\frac{[\text{CR}_{\text{ES}}] \quad \Gamma \vdash (\nu \tilde{s})P \rhd_{\tilde{t}\setminus\tilde{s}} \Delta}{\Gamma \vdash (\nu \tilde{s})P \rhd_{\tilde{t}\setminus\tilde{s}} \Delta} \right]\!\!\right]$$
$$= (\nu \tilde{s}, \mathtt{in}_{\tilde{s}1}, \mathtt{out}_{\tilde{s}1}, \ldots, \mathtt{in}_{\tilde{s}n}, \mathtt{out}_{\tilde{s}n})(\mathcal{C}[\![G]\!]^{\star}_{\tilde{s},n} \mid \mathtt{in}_{\tilde{s}1} : \varnothing \mid \ldots \mid \mathtt{out}_{\tilde{s}n} : \varnothing \mid \mathcal{E}[\![\mathcal{D}_1)]\!]$$

Monomorphically for all other cases.

Figure 69 Illustration of the completeness proof

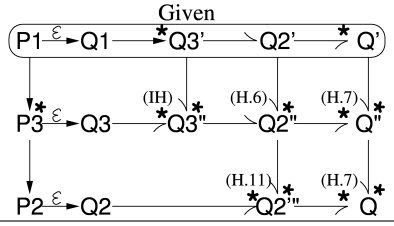


Figure 70 Example of partially specialised nurse process

```
P_N =
a[3](d,s,r). if (busy)
then sync((d,s,r),3) {
  CaseD: sync((d,s,r),3) { CaseDD: end, CaseDN: s!⟨eSchedule⟩; end },
  CaseN: d?(data); sync((d,s,r),3) { CaseND: end } }
else sync((d,s,r),3) {
  CaseD: sync((d,s,r),3) { CaseDN: s!⟨eSchedule⟩; end },
  CaseN: d?(data); sync((d,s,r),3) { CaseND: end, CaseNN: s!⟨eSchedule⟩; end } }
```

Figure 72

```
G=
{CaseD:
  1→2:1⟨Sdata⟩;
  {CaseDD:
    2→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end,
   CaseDN:
    3→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end
  },
 CaseN:
  1→3:1⟨Sdata⟩;
  {CaseND:
    2→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end,
   CaseNN:
    3→1:2⟨Sschedule⟩;2→1:3⟨Sresult⟩;end
  }
}

G↾2=
{ CaseD:
  1?⟨Sdata⟩;
  { CaseDD:
    2!⟨Sschedule⟩;3!⟨Sresult⟩;end,
   CaseDN:
    3!⟨Sresult⟩;end
  },
  CaseN:
  { CaseND:
    2!⟨Sschedule⟩;3!⟨Sresult⟩;end,
   CaseNN:
    3!⟨Sresult⟩;end
  }
}

G↾1=
{CaseD:
  1!⟨Sdata⟩;
  {CaseDD:
    2?⟨Sschedule⟩;3?⟨Sresult⟩;
      end,
   CaseDN:
    2?⟨Sschedule⟩;3?⟨Sresult⟩;
      end
  },
 CaseN:
  1!⟨Sdata⟩;
  {CaseND:
    2?⟨Sschedule⟩;3?⟨Sresult⟩;
      end,
   CaseNN:
    2?⟨Sschedule⟩;3?⟨Sresult⟩;
      end
  }
}

G↾3=
{ CaseD:
  { CaseDD:
    end,
   CaseDN:
    2!⟨Sschedule⟩;end
  },
  CaseN:
  1?⟨Sdata⟩;
  { CaseND:
    end,
   CaseNN:
    2!⟨Sschedule⟩;end
  }
}
```

Figure 73 Global type representing the process matrix from Figure 50

```
{ Pdata:
  1→2:2⟨String⟩;
  1→3:3⟨String⟩;
  μ stateD.
  { Pdata:
    1→2:2⟨String⟩;
    1→3:3⟨String⟩;
    stateD,
    Dschedule:
    2→1:1⟨String⟩;
    2→3:2⟨String⟩;
    μ stateDS.
    { Pdata:
      1→2:2⟨String⟩;
      1→3:3⟨String⟩;
      stateD,
      Dschedule:
      2→1:1⟨String⟩;
      2→3:3⟨String⟩;
      stateDS,
      Nschedule:
      3→1:1⟨String⟩;
      3→2:2⟨String⟩;
      stateDS,
      Dresult:
      2→1:1⟨String⟩;
    end
  },
  Nschedule:
  3→1:1⟨String⟩;
  3→2:2⟨String⟩;
  μ stateDS.
  { Pdata:
    1→2:2⟨String⟩;
    1→3:3⟨String⟩;
    stateD,
    Dschedule:
    2→1:1⟨String⟩;
    2→3:3⟨String⟩;
    stateDS,
    Nschedule:
    3→1:1⟨String⟩;
    3→2:2⟨String⟩;
    stateDS,
    Dresult:
    2→1:1⟨String⟩;
    end
  }
}
```

Figure 74 Local type for the doctor

```
{ Pdata:
  2?⟨String⟩;
  μ stateD.
  { Pdata:
    2?⟨String⟩;
    stateD,
    Dschedule:
    1!⟨String⟩;
    2!⟨String⟩;
    μ stateDS.
    { Pdata:
      2?⟨String⟩;
      stateD,
      Dschedule:
      1!⟨String⟩;
      3!⟨String⟩;
      stateDS,
      Nschedule:
      2?⟨String⟩;
      stateDS,
      Dresult:
      1!⟨String⟩;
    end
  },
  Nschedule:
  2?⟨String⟩;
  μ stateDS.
  { Pdata:
    2?⟨String⟩;
    stateD,
    Dschedule:
    1!⟨String⟩;
    3!⟨String⟩;
    stateDS,
    Nschedule:
    2?⟨String⟩;
    Dresult:
    1!⟨String⟩;
    end
  }
}
```

D

MULTIPARTY SYMMETRIC SUM TYPES WITH ASSERTIONS

D.1 DEFINITIONS

D.1.1 *Process congruence*

The process semantics uses the notion of process equivalence ($\equiv$), and this is included in Figure 75.

Figure 75 Process congruence ($\equiv$)

The relation $\equiv$ is defined as the smallest *congruence* relation satisfying

$P|0 \equiv P$ $\qquad$ $P|Q \equiv Q|P$

$P|(Q|R) \equiv (P|Q)|R$ $\qquad$ $(\nu n)P|Q \equiv (\nu n)(P|Q)$ if $n \notin \mathsf{fn}(Q)$

$(\nu n n')P \equiv (\nu n' n)P$ $\qquad$ $(\nu n)0 \equiv 0$

$\mathsf{def}\ D\ \mathsf{in}\ 0 \equiv 0$ $\qquad$ $(\nu s_1..s_n)\Pi_i s_i : \varnothing \equiv 0$

$\mathsf{def}\ D\ \mathsf{in}\ (\nu n)P \equiv (\nu n)\mathsf{def}\ D\ \mathsf{in}\ P$ if $n \notin \mathsf{fn}(D)$

$(\mathsf{def}\ D\ \mathsf{in}\ P) \mid Q \equiv \mathsf{def}\ D\ \mathsf{in}\ (P \mid Q)$ if $\mathsf{dpv}(D) \cap \mathsf{fpv}(Q) = \varnothing$

$\mathsf{def}\ D\ \mathsf{in}\ \mathsf{def}\ D'\ \mathsf{in}\ P \equiv \mathsf{def}\ D\ \mathsf{and}\ D'\ \mathsf{in}\ P$ if $\mathsf{dpv}(D) \cap \mathsf{dpv}(D') = \varnothing$

D.1.2 *Symmetric sum types*

We include the full set of typing rules in Figure 43. The typing system uses the notion of type projection ($\upharpoonright$) defined in Figure 64, and the notion of subtyping ($\leqslant$) of session-environments which consists of subtyping of each of the used local types defined in Figure 65.

The expression typing rules ($\Gamma \vdash e : S$)

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ [\textsc{True}] \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ [\textsc{False}] \qquad \frac{}{\Gamma, x : S \vdash x : S}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \mathsf{not}\ e : \mathsf{bool}}\ [\textsc{Not}]$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1\ \mathsf{and}\ e_2 : \mathsf{bool}}\ [\textsc{And}] \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1\ \mathsf{or}\ e_2 : \mathsf{bool}}\ [\textsc{Or}] \qquad \cdots$$

The typing rules ($\Theta; \Gamma \vdash P \triangleright \Delta$)

$$\frac{[\textsc{Subs}] \quad \Theta; \Gamma \vdash P \triangleright \Delta \quad \vdash \Theta \Rightarrow \Delta \leqslant \Delta'}{\Theta; \Gamma \vdash P \triangleright \Delta'}$$

$$\frac{\begin{array}{c}\forall l \in L'' : \Theta \wedge A_l; \Gamma \vdash P_l \triangleright \Delta, \tilde{s} : T_l@(p,n) \quad L'' \subseteq L \cup L' \\ \forall l \in L \backslash L'' :\vdash \Theta \Rightarrow \neg B_l \quad \forall l \in L'' :\vdash \Theta \Rightarrow (A_l \Rightarrow B_l) \\ [\textsc{Sync}] \qquad \forall l \in L :\vdash \Theta \Rightarrow (B_l \Rightarrow A_l) \quad \vdash \Theta \Rightarrow \bigvee_{l \in L} B_l\end{array}}{\Theta; \Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{\{A_l\}\, l : P_l\}_{l \in L''} \triangleright \Delta, \tilde{s} : \{\{B_l\}\, l : T_l\}_{l \in L;L'}@(p,n)}$$

$$\frac{\begin{array}{rl} & |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ [\textsc{Mcast}] \quad \Gamma \vdash a : \langle G \rangle \quad \Theta; \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright 1)@(1,n) & n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta; \Gamma \vdash \overline{a}[2..n](\tilde{s}).P \triangleright \Delta}$$

$$\frac{\begin{array}{rl} & |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ [\textsc{Macc}] \quad \Gamma \vdash a : \langle G \rangle \quad \Theta; \Gamma \vdash P \triangleright \Delta, \tilde{s} : (G \upharpoonright p)@(p,n) & n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta; \Gamma \vdash a[p](\tilde{s}).P \triangleright \Delta}$$

$$\frac{[\textsc{SendA}] \quad \Gamma \vdash e : S \quad \Theta; \Gamma \vdash P \triangleright \Delta, s : T[e/x]@(p,n) \quad \vdash \Theta \Rightarrow A[e/x]}{\Theta; \Gamma \vdash s_k!\langle e \rangle; P \triangleright \Delta, \tilde{s} : k!\langle S \rangle \text{ as } x\{A\}; T@(p,n)}$$

$$\frac{[\textsc{RcvA}] \qquad \Theta \wedge A; \Gamma, x : S \vdash P \triangleright \Delta, \tilde{s} : T@(p,n)}{\Theta; \Gamma \vdash s_k?(x); P \triangleright \Delta, \tilde{s} : k?\langle S \rangle \text{ as } x\{A\}; T@(p,n)} \; (x \notin fv(\Theta) \cup fv(\Delta))$$

$$\frac{[\textsc{Send}] \quad \forall j . \Gamma \vdash e_j : S_j \quad \Theta; \Gamma \vdash P \triangleright \Delta, s : T@(p,n)}{\Theta; \Gamma \vdash s_k!\langle \tilde{e} \rangle; P \triangleright \Delta, \tilde{s} : k!\langle \tilde{S} \rangle; T@(p,n)}$$

$$\frac{[\textsc{Rcv}] \quad \Theta; \Gamma, \tilde{x} : \tilde{S} \vdash P \triangleright \Delta, \tilde{s} : T@(p,n)}{\Theta; \Gamma \vdash s_k?(\tilde{x}); P \triangleright \Delta, \tilde{s} : k?\langle \tilde{S} \rangle; T@(p,n)} \; (\tilde{x} \cap (fv(\Theta) \cup fv(\Delta)) = \varnothing)$$

$$\frac{[\textsc{If}] \quad \Gamma \vdash e : \mathsf{bool} \quad \Theta \wedge e; \Gamma \vdash P \triangleright \Delta \quad \Theta \wedge \neg e; \Gamma \vdash Q \triangleright \Delta}{\Theta; \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta}$$

$$\frac{[\textsc{Deleg}] \qquad \Theta; \Gamma \vdash P \triangleright \Delta, \tilde{s} : T@(p,n)}{\Theta; \Gamma \vdash s_k!\langle\!\langle \tilde{t} \rangle\!\rangle; P \triangleright \Delta, \tilde{s} : k!\langle T'@(p', |\tilde{t}|, n') \rangle; T@(p,n), \tilde{t} : T'@(p', n')}$$

$$\frac{[\textsc{Srec}] \qquad \Theta; \Gamma \vdash P \triangleright \Delta, \tilde{s} : T@(p,n), \tilde{t} : T'@(p', n')}{\Theta; \Gamma \vdash s_k?((\tilde{t})); P \triangleright \Delta, \tilde{s} : k?\langle T'@(p', |\tilde{t}|, n') \rangle; T@(p,n)}$$

$$\frac{[\textsc{Sel}] \quad \Theta; \Gamma \vdash P \triangleright \Delta, \tilde{s} : T@(p,n) \quad h \in L \quad \vdash \Theta \Rightarrow A_h}{\Theta; \Gamma \vdash s_k \lhd h; P \triangleright \Delta, \tilde{s} : k \oplus \{\{A_l\}\, l : T_l\}_{l \in L}@(p,n)}$$

$$\frac{[\textsc{Branch}] \qquad \forall l \in L : \Theta \wedge A_l; \Gamma \vdash P_l \triangleright \Delta, \tilde{s} : T_l@(p,n)}{\Theta; \Gamma \vdash s_k \rhd \{l : P_l\}_{l \in L} \triangleright \Delta, \tilde{s} : k\&\{\{A_l\}\, l : T_l\}_{l \in L}@(p,n)}$$

$$\frac{[\textsc{Conc}] \quad \Theta;\Gamma \vdash P \triangleright \Delta \quad \Theta;\Gamma \vdash Q \triangleright \Delta'}{\Theta;\Gamma \vdash P|Q \triangleright \Delta \circ \Delta'} \ (\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \varnothing)$$

$$\frac{[\textsc{Inact}] \quad \Delta \quad \text{end only}}{\Theta;\Gamma \vdash 0 \triangleright \Delta} \qquad \frac{[\textsc{Nres}] \quad \Theta;\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Theta;\Gamma \vdash (\nu a)P \triangleright \Delta}$$

$$\frac{[\textsc{Var}] \qquad \forall j.\Gamma \vdash e_j : S_j \quad \Delta \quad \text{end only}}{\Theta;\Gamma, X : (\tilde{x} : \tilde{S})T@(\tilde{p}, n) \vdash X\langle \tilde{e} \rangle(\tilde{s}_1 ... \tilde{s}_{|\tilde{T}|}) \triangleright \Delta, \tilde{T}[\tilde{e}/\tilde{x}]@(p, n)}$$

$$\frac{[\textsc{Def}] \quad \Theta;\Gamma, X : (\tilde{x} : \tilde{S})T@(\tilde{p}, n), \tilde{x} : \tilde{S} \vdash P \triangleright T@(\tilde{p}, n) \quad \Theta;\Gamma, X : (\tilde{x} : \tilde{S})T@(\tilde{p}, n) \vdash Q \triangleright \Delta}{\Theta;\Gamma \vdash \mathsf{def}\ X\langle \tilde{x} \rangle(\tilde{s}_1, ..., \tilde{s}_{|\tilde{T}|}) = P\ \mathsf{in}\ Q \triangleright \Delta}$$

The runtime typing rules ($\Theta;\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$)

Where Δ in the static typing rules represents a map from $\tilde{s}$ to $T@(p, n)$, the runtime typing rules uses Δ as a map from $(\tilde{s}, p)$ to $T@(p, n)$.

The extra $\tilde{t}$ is used to ensure that there is exactly one queue for each session channel in each open session.

$$\frac{[\textsc{Subs}] \quad \Theta;\Gamma \vdash P \triangleright_{\tilde{t}} \Delta \quad \vdash \Theta \Rightarrow \Delta \leqslant \Delta'}{\Theta;\Gamma \vdash P \triangleright_{\tilde{t}} \Delta'}$$

$$\frac{\begin{array}{c}\forall l \in L'' : \Theta \wedge A_l;\Gamma \vdash P_l \triangleright_{\tilde{t}} \Delta, \tilde{s} : T_l@(p, n) \quad L'' \subseteq L \cup L' \\ \forall l \in L \backslash L'' :\vdash \Theta \Rightarrow \neg B_l \quad \forall l \in L'' :\vdash \Theta \Rightarrow (A_l \Rightarrow B_l) \\ [\textsc{Sync}] \qquad \forall l \in L :\vdash \Theta \Rightarrow (B_l \Rightarrow A_l) \quad \vdash \Theta \Rightarrow \bigvee_{l \in L} B_l\end{array}}{\Theta;\Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{\{A_l\}\, l : P_l\}_{l \in L''} \triangleright_{\tilde{t}} \Delta, \tilde{s} : \{\{B_l\}\, l : T_l\}_{l \in L;L'}@(p, n)}$$

$$\frac{[\textsc{Mcast}] \quad \Gamma \vdash a : \langle G \rangle \quad \Theta;\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, \tilde{s} : (G \upharpoonright 1)@(1, n) \quad \begin{array}{l}|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta;\Gamma \vdash \overline{a}[2..n](\tilde{s}).P \triangleright_{\tilde{t}} \Delta}$$

$$\frac{[\textsc{Macc}] \quad \Gamma \vdash a : \langle G \rangle \quad \Theta;\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, \tilde{s} : (G \upharpoonright p)@(p, n) \quad \begin{array}{l}|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta;\Gamma \vdash a[p](\tilde{s}).P \triangleright_{\tilde{t}} \Delta}$$

$$\frac{[\textsc{SendA}] \quad \Gamma \vdash e : S \quad \Theta;\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, s : T[e/x]@(p, n) \quad \vdash \Theta \Rightarrow A[e/x]}{\Theta;\Gamma \vdash s_k!\langle e \rangle; P \triangleright_{\tilde{t}} \Delta, \tilde{s} : k!\langle S \rangle\ \mathsf{as}\ x\,\{A\}\,; T@(p, n)}$$

$$\frac{[\textsc{RcvA}] \qquad \Theta \wedge A;\Gamma, x : S \vdash P \triangleright_{\tilde{t}} \Delta, \tilde{s} : T@(p, n)}{\Theta;\Gamma \vdash s_k?(x); P \triangleright_{\tilde{t}} \Delta, \tilde{s} : k?\langle S \rangle\ \mathsf{as}\ x\,\{A\}\,; T@(p, n)} \ (x \notin fv(\Theta) \cup fv(\Delta))$$

$$\frac{[\textsc{Send}] \quad \forall j.\Gamma \vdash e_j : S_j \quad \Theta;\Gamma \vdash P \triangleright_{\tilde{t}} \Delta, s : T@(p, n)}{\Theta;\Gamma \vdash s_k!\langle \tilde{e} \rangle; P \triangleright_{\tilde{t}} \Delta, \tilde{s} : k!\langle \tilde{S} \rangle; T@(p, n)}$$

$$\frac{[\text{Rcv}] \quad \Theta; \Gamma, \tilde{x} : \tilde{S} \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(p, n)}{\Theta; \Gamma \vdash s_k?(\tilde{x}); P \rhd_{\tilde{t}} \Delta, \tilde{s} : k?\langle \tilde{S} \rangle; T@(p, n)} \ (\tilde{x} \cap (fv(\Theta) \cup fv(\Delta)) = \varnothing)$$

$$\frac{[\text{If}] \quad \Gamma \vdash e : \text{bool} \quad \Theta \wedge e; \Gamma \vdash P \rhd_{\tilde{t}} \Delta \quad \Theta \wedge \neg e; \Gamma \vdash Q \rhd_{\tilde{t}} \Delta}{\Theta; \Gamma \vdash \text{if } e \text{ then } P \text{ else } Q \rhd_{\tilde{t}} \Delta}$$

$$\frac{[\text{Deleg}] \qquad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(p, n)}{\Theta; \Gamma \vdash s_k!\langle\!\langle \tilde{t} \rangle\!\rangle; P \rhd_{\tilde{t}} \Delta, \tilde{s} : k!\langle T'@(p', |\tilde{t}|, n') \rangle; T@(p, n), \tilde{t} : T'@(p', n')}$$

$$\frac{[\text{Srec}] \qquad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(p, n), \tilde{t} : T'@(p', n')}{\Theta; \Gamma \vdash s_k?((\tilde{t})); P \rhd_{\tilde{t}} \Delta, \tilde{s} : k?\langle T'@(p', |\tilde{t}|, n') \rangle; T@(p, n)}$$

$$\frac{[\text{Sel}] \quad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(p, n) \quad h \in L \quad \vdash \Theta \Rightarrow A_h}{\Theta; \Gamma \vdash s_k \lhd h; P \rhd_{\tilde{t}} \Delta, \tilde{s} : k \oplus \{\{A_l\}\, l : T_l\}_{l \in L}@(p, n)}$$

$$\frac{[\text{Branch}] \qquad \forall l \in L : \Theta \wedge A_l; \Gamma \vdash P_l \rhd_{\tilde{t}} \Delta, \tilde{s} : T_l@(p, n)}{\Theta; \Gamma \vdash s_k \rhd \{l : P_l\}_{l \in L} \rhd_{\tilde{t}} \Delta, \tilde{s} : k\&\{\{A_l\}\, l : T_l\}_{l \in L}@(p, n)}$$

$$\frac{[\text{Conc}] \quad \Theta; \Gamma \vdash P \rhd_{\tilde{t}_1} \Delta \quad \Theta; \Gamma \vdash Q \rhd_{\tilde{t}_1} \Delta' \quad \Delta \asymp \Delta' \quad \tilde{t}_1 \cap \tilde{t}_2 = \varnothing}{\Theta; \Gamma \vdash P | Q \rhd_{\tilde{t}_1 \cup \tilde{t}_2} \Delta \circ \Delta'}$$

$$\frac{[\text{Inact}] \quad \Delta \quad \text{end only}}{\Theta; \Gamma \vdash 0 \rhd_{\tilde{t}} \Delta} \qquad \frac{[\text{Nres}] \quad \Theta; \Gamma, a : \langle G \rangle \vdash P \rhd_{\tilde{t}} \Delta}{\Theta; \Gamma \vdash (\nu a) P \rhd_{\tilde{t}} \Delta}$$

$$\frac{[\text{Cres}] \quad \Theta; \Gamma, \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : \{T_p@(p, n)\}_{p=1}^{n} \quad \{T_p@(p, n)\}_{p=1}^{n} \text{ coherent} \quad \tilde{s} \subseteq \tilde{t}}{\Theta; \Gamma \vdash (\nu \tilde{s}) P \rhd_{\tilde{t} \setminus \tilde{s}} \Delta}$$

$$\frac{[\text{Var}] \qquad \forall j . \Gamma \vdash e_j : S_j \quad \Delta \quad \text{end only}}{\Theta; \Gamma, X : (\tilde{x} : \tilde{S}) T@(\tilde{p}, n) \vdash X\langle \tilde{e} \rangle(\tilde{s}_1 ... \tilde{s}_{|\tilde{T}|}) \rhd_{\tilde{t}} \Delta, \tilde{T}[\tilde{e}/\tilde{x}]@(p, n)}$$

$$\frac{[\text{Def}] \quad \Theta; \Gamma, X : (\tilde{x} : \tilde{S}) T@(\tilde{p}, n), \tilde{x} : \tilde{S} \vdash P \rhd_{\varnothing} T@(\tilde{p}, n) \quad \Theta; \Gamma, X : (\tilde{x} : \tilde{S}) T@(\tilde{p}, n) \vdash Q \rhd_{\tilde{t}} \Delta}{\Theta; \Gamma \vdash \text{def } X\langle \tilde{x} \rangle(\tilde{s}_1, ..., \tilde{s}_{|\tilde{T}|}) = P \text{ in } Q \rhd_{\tilde{t}} \Delta}$$

Plus queue rules

D.2 SUBJECT REDUCTION

We include the proof of subject reduction in Proof D.2.2. The proof uses Lemma D.2.1 which is an extension of Lemma 5.18 from [61]. It states that the [Subs] rules can be propagated upwards to the [Sel] and [Branch] rules.

Lemma D.2.1 (Extension of Lemma 5.18 from [61]: Permutation)**.**

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta'}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta''} \quad \textit{then} \quad \dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta''}$$

(1) *If* and *then* .

$$\dfrac{\dfrac{[X] \quad \mathcal{D}}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta}\,[\textsc{Subs}]}{\Gamma \vdash P \rhd_{\tilde{t}} \Delta'}$$

(2) *If* *and the second last rule-application X is not Sel or Branch then the last two rule-applications can be permuted.*

PROOF:

(1) Is immediate because $\leqslant_{\text{sub}}$ is transitive.

(2) Is proved for each possible rule X. This is done as in the original proof. There is one new case, and we will prove it now.

Sync: In this case we consider a derivation

$$\dfrac{\dfrac{\forall l \in L'' \quad \dfrac{\mathcal{D}_l}{\Theta \wedge A_l; \Gamma \vdash P_l \rhd_{\tilde{t}} \Delta, \tilde{s} : \{T_l@(p,n)\}}\,[\textsc{Sync}] \quad \cdots}{\Theta; \Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{\{B_l\}\, l : P_l\}_{l \in L''} \rhd_{\tilde{t}} \Delta, \tilde{s} : \{\{\{A_l\}\, l : T_l\}_{l \in L;L'}@(p,n)\}}\,[\textsc{Subs}]}{\Theta; \Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{\{B_l\}\, l : P_l\}_{l \in L''} \rhd_{\tilde{t}} \Delta', \tilde{s} : \{\{\{A_l\}\, l : T'_l\}_{l \in L;L'}@(p,n)\}}$$

where $T_l \leqslant_{\text{sub}} T'_l$ for each $l \in L''$ and $\Delta \leqslant_{\text{sub}} \Delta'$. We can therefore create

$$\dfrac{\forall l \in L'' \quad \dfrac{\dfrac{\mathcal{D}_l}{\Gamma \vdash P_l \rhd_{\tilde{t}} \Delta, \tilde{s} : \{T_l@(p,n)\}}\,[\textsc{Subs}]}{\Theta \wedge A_l; \Gamma \vdash l : P_l \rhd_{\tilde{t}} \Delta', \tilde{s} : \{T'_l@(p,n)\}} \quad \cdots}{\Theta; \Gamma \vdash \mathsf{sync}_{\tilde{s},n}\{\{B_l\}\, l : P_l\}_{l \in L''} \rhd_{\tilde{t}} \Delta', \tilde{s} : \{\{\{A_l\}\, l : T'_l\}_{l \in L;L'}@(p,n)\}}\,[\textsc{Sync}]$$

□

Proof D.2.2 (Theorem 5.3.2: Subject Reduction)**.**
We prove

> *If* $\mathsf{true}; \Gamma \vdash P \rhd_{\tilde{s}} \Delta$, Δ *coherent and* $P \to P'$
> *then* $\mathsf{true}; \Gamma \vdash P' \rhd_{\tilde{s}} \Delta'$*where* $\Delta \to^{0/1} \Delta'$.

By induction on the derivation of $P \to P'$.
We prove the case Sync. The remaining cases can be generated by adding assertion arguments to the proof in [61]. Assume

$$\dfrac{h \in \bigcap_{i=1}^{n} L_i \quad B_{1h} \downarrow \mathsf{true} \quad \ldots \quad B_{nh} \downarrow \mathsf{true}}{\mathit{sync}_{\tilde{s},n}\{\{B_{1l}\}\, l : P_{1l}\}_{l \in L_1} \mid \ldots \mid \mathit{sync}_{\tilde{s},n}\{\{B_{nl}\}\, l : P_{nl}\}_{l \in L_n} \to P_{1h} \mid \ldots \mid P_{nh}}\,[\textsc{Sync}]$$

We can assume that the typing $\mathsf{true}; \Gamma \vdash \mathit{sync}_{\tilde{t},n}\{\{B_{1l}\}\, l : P_{1l}\}_{l\in L_1} \mid \ldots \mid \mathit{sync}_{\tilde{t},n}\{\{B_{nl}\}\, l : P_{nl}\}_{l\in L_n} \rhd_{\tilde{s}} \Delta, \tilde{t} : \{\{\{A_{il}\}\, l : T_{il}\}_{l\in L;L'}@(i,n)\}_{i\in\{1..n\}}$ *starts with* $n-1$ *applications of the Conc rule each containing one application of the Sync rule because of the extension of Lemma 5.18 in D.2.1. This gives us the subderivations:*

$$\dfrac{\dfrac{\mathcal{D}_{il}}{\forall l \in L_i.\ \mathsf{true} \wedge A_{il}; \Gamma \vdash P_{il} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{T_{il}@(i,n)\}} \quad \forall l \in L_i.\ \vdash \mathsf{true} \Rightarrow (B_{il} \Rightarrow A_{il}) \quad \cdots}{\mathsf{true}; \Gamma \vdash \mathit{sync}_{\tilde{t},n}\{\{B_{il}\}\, l : P_{il}\}_{l\in L_i} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{\{A_{il}\}\, l : T_{il}\}_{l\in L';L}@(i,n)}\ [\textsc{Sync}]$$

for i*=1..*n *such that* $\tilde{s}_i \cap \tilde{s}_j = \varnothing$ *for all* $i \neq j$ *in* $1..n$, $\bigcup_{i=1}^{n} \tilde{s}_i = \tilde{s}$ *and* $\Delta_1 \circ (\Delta_2 \circ (\ldots \circ \Delta_n)) = \Delta$.

Since each of these subderivations starts with the Sync rule we get that

$$\dfrac{\mathcal{D}_{ih}}{\mathsf{true} \wedge A_{ih}; \Gamma \vdash P_{ih} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{T_{ih}@(i,n)\}} \quad \textit{for } i = 1..n$$

Since for each i: $B_{ih} \downarrow \mathsf{true}$ *we have that* $\vdash B_{ih}$*, and since* $\vdash \mathsf{true} \Rightarrow (B_{ih} \Rightarrow A_{ih})$ *we get that* $\vdash A_{ih}$ *by cut elimination of* B_{ih}*. Finaly by applying cut-elimination of* A_{ih} *to each proof in* $\mathcal{D}_{ih}$ *we get that*

$$\dfrac{\mathcal{D}'_{ih}}{\mathsf{true}; \Gamma \vdash P_{ih} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{T_{ih}@(i,n)\}} \quad \textit{for } i = 1..n$$

Now we can apply Conc $n-1$ *times to create a derivation of*
$\mathsf{true}; \Gamma \vdash P_{1h} \mid \ldots \mid P_{nh} \rhd_{\tilde{s}} \Delta, \tilde{t} : \{T_{ih}@(p,n)\}_{p\in\{1..n\}}$.
Since $\Delta, \tilde{t} : \{\{\{A_{il}\}\, l : T_{il}\}_{l\in L;L'}@(i,n)\}_{i\in\{1..n\}} \to \Delta, \tilde{t} : \{T_{ih}@(i,n)\}_{i\in\{1..n\}}$*, subject reduction is fulfilled in the Sync case.* □

BIBLIOGRAPHY

[1] Apims project page. http://www.thelas.dk/index.php/apims.

[2] Boost.join: C++ asynchronous message coordination and concurrency library. http://channel.sourceforge.net/.

[3] The danish patient security database. http://www.dpsd.dk/.

[4] Parallel c#. http://www.parallelcsharp.com/.

[5] *Patientsikkerhed i primærsektoren – eksempler på utilsigtede hændelser*. Dansk Selskab for Patientsikkerhed, 2010.

[6] Business process model and notation (bpmn) version 2. http://www.omg.org/spec/BPMN/2.0/PDF, 2011.

[7] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proc. 1996 IEEE 11th Annual Symp. on Logic in Computer Science (LICS), New Brunswick, New Jersey*. IEEE Computer Society Press, June 1996.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: principles, techniques, and tools. *Reading, MA,*, 1986.

[9] Alexandre Alves, Assaf Arkin, Sid Askary abd Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Ws-bpel oasis web services business process execution language. URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

[10] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, September 1993.

[11] Valentin Antimirov. Rewriting regular inequalities. In *Proc. 10th International Conference, FCT '95 Dresden, Germany*, volume 965 of *Lecture Notes in Computer Science (LNCS)*, pages 116–125. Springer-Verlag, August 1995.

[12] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(95)00182-4.

[13] Valentin M. Antimirov and Peter D. Mosses. Rewriting extended regular expressions. *Theor. Comput. Sci.*, 143(1):51–72, 1995. doi: http://dx.doi.org/10.1016/0304-3975(95)80024-4.

[14] Chagit Attiya, Danny Dolev, and Joseph Gil. Asynchronous byzantine consensus. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 119–133, New York, NY, USA, 1984. ACM. ISBN 0-89791-143-1. doi: http://doi.acm.org/10.1145/800222.806740.

[15] Joshua S. Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *ICDCS*, pages 393–402, 1999.

[16] J. E. Bardram. Activity-based computing for medical work in hospitals. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(2):10, 2009.

[17] Lorenzo Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.

[18] P. Bille and M. Thorup. Faster regular expression matching. *Automata, Languages and Programming*, pages 171–182, 2009.

[19] OpenMP Architecture Review Board. Openmp. `http://www.openmp.org`. URL `www.openmp.org`.

[20] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. *CONCUR 2010-Concurrency Theory*, pages 162–176, 2011.

[21] Eduardo Bonelli and Adriana B. Compagnoni. Multipoint session types for a distributed calculus. In *TGC*, volume 4912 of *LNCS*, pages 240–256. Springer, 2007.

[22] Claus Brabrand and Jakob Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. In *Proc. 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2010.

[23] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998.

[24] D. Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, 191(1-2):131–144, 1998.

[25] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321239.321249.

[26] Luís Caires and Hugo Torres Vieira. Conversation types. In *ESOP '09*, pages 285–300, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-00589-3. doi: http://dx.doi.org/10.1007/978-3-642-00590-9_21.

[27] R. D. Cameron. Source encoding using syntactic information source models. *Information Theory, IEEE Transactions on*, 34(4):843–850, 1988. ISSN 0018-9448.

[28] Giuseppe Castagna and Luca Padovani. Contracts for mobile processes. In *CONCUR '09*, LNCS, pages 211–228, Berlin, Heidelberg, 2009. Springer.

[29] Hubie Chen and Riccardo Pucella. A coalgebraic approach to kleene algebra with tests. *Theor. Comput. Sci.*, 327(1-2):23–44, 2004.

[30] N. Chomsky. On certain formal properties of grammars*. *Information and control*, 2(2):137–167, 1959.

[31] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for objective-caml. In *Agent Systems and Applications, 1999 and Third International Symposium on Mobile Agents. Proceedings. First International Symposium on*, pages 22–29, 1999.

[32] J. F. Contla. Compact coding of syntactically correct source programs. *Software: Practice and Experience*, 15(7):625–636, 1985. ISSN 1097-024X.

[33] J. H. Conway. *Regular Algebra and Finite Machines*. Printed in GB by William Clowes & Sons Ltd, 1971. ISBN 0-412-10620-5.

[34] Russ Cox. Regular expression matching can be simple and fast. `http://swtch.com/~rsc/regexp/regexp1.html`. URL `http://swtch.com/~rsc/regexp/regexp1.html`.

[35] Roberto Di Cosmo, Francois Pottier, and Didier Remy. Subtyping recursive types modulo associative commutative products. In *Proc. Seventh International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, 2005.

[36] Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, September 2000. doi: 10.1007/s002360000037.

[37] Institute of Electrical and Electronics Engineers (IEEE). *Standard for information technology — Portable Operating System Interface (POSIX) — Part 2 (Shell and utilities), Section 2.8 (Regular expression notation)*. New York, 1992. IEEE Standard 1003.2.

[38] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. *SIGPLAN Not.*, 39(1):77–88, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/982962.964008.

[39] Marcelo P. Fiore. A coinduction principle for recursive data types based on bisimulation. *Information and Computation*, 127:186–198, 1996. Conference version: Proc. 8th Annual IEEE Symp. on Logic in Computer Science (LICS), 1993, pp. 110-119.

[40] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 127–140, London, UK, 1983. Springer. ISBN 3-540-12689-9.

[41] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/3149.214121.

[42] The Royal Dutch Society for Physical Therapy (KNGF). Kngf evidence-based clinical practice guidelines. `https://www.kngfrichtlijnen.nl/654/KNGF-Guidelines-in-English.htm`. URL `https://www.kngfrichtlijnen.nl/654/KNGF-Guidelines-in-English.htm`.

[43] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Implicit typing à la ml for the join-calculus. *CONCUR'97: Concurrency Theory*, pages 196–212, 1997.

[44] J. Fox, N. Johns, and A. Rahmanzadeh. Disseminating medical knowledge: the proforma approach. *Artificial Intelligence in Medicine*, 14(1-2):157–182, 1998.

[45] N. Freemantle, E. L. Harvey, F. Wolf, J. M. Grimshaw, R. Grilli, and L. A. Bero. Printed educational materials: effects on professional practice and health care outcomes. *Cochrane Database Syst Rev*, 2, 2000.

[46] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture notes in computer science*, pages 618–629, Turku, Finland, July 2004. Springer.

[47] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. *J. Funct. Program.*, 12(6):511–548, 2002.

[48] S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, 2005.

[49] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[50] A. Ginzburg. A procedure for checking equality of regular expressions. *J. ACM*, 14(2):355–362, 1967. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321386.321399.

[51] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. In *CONCUR '08*, LNCS, pages 492–507, 2008.

[52] Clemens Grabmayer. Using proofs by coinduction to find "traditional" proofs. In *Proc. 1st Conference on Algebra and Coalgebra in Computer Science (CALCO)*, number 3629 in Lecture Notes in Computer Science (LNCS). Springer, September 2005.

[53] A. B. Haynes, T. G. Weiser, W. R. Berry, S. R. Lipsitz, A. H. Breizat, E. P. Dellinger, T. Herbosa, S. Joseph, P. L. Kibatala, M. C. Lapitan, et al. A surgical safety checklist to reduce morbidity and mortality in a global population. *N Engl J Med*, 360(5):491–499, 2009.

[54] R. Hempel. The mpi standard for message passing. In *High-Performance Computing and Networking*, pages 247–252, 1994.

[55] Fritz Henglein and Lasse Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation (preliminary version). Technical report, Department of Computer Science, University of Copenhagen (DI, February 2010. D-612.

[56] Fritz Henglein and Lasse Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation. In *Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2011.

[57] A. S. Henriksen. A contraction-free focused sequent calculus for classical propositional logic. Technical report, `http://www.diku.dk/hjemmesider/ansatte/starcke/`, 2010.

[58] Thomas Hildebrandt, Karen Lyng, and Raghava Mukkamala. From paper based clinical practice guidelines to declarative workflow and linear-time temporal logic. 2009.

[59] Tony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[60] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[61] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

[62] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[63] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for xml. In Jens Palsberg and Martín Abadi, editors, *POPL*, Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, pages 50–62. ACM, 2005. ISBN 1-58113-830-X.

[64] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[65] G. Hripcsak, P. D. Clayton, and T. Pryor. The arden syntax for medical logic modules. In *14. Annual Symposium on Computer Applications in Medical Care*, pages 200–204, 1990.

[66] Raymond Hu, Nobuko Yoshida, Andi Bejleri, and Kohei Honda. The sj framework for transport-independent, type-safe, object-oriented communications programming. 2009. URL http://www.doc.ic.ac.uk/~rhu/sessionj-ti.html.

[67] Intalio. Intalio|bpmn. http://www.intalio.com/bpms.

[68] P. Jansson and J. Jeuring. Polyp—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, page 482, 1997.

[69] P. Jansson and J. Jeuring. Polytypic compact printing and parsing. *Programming Languages and Systems*, pages 639–639, 1999.

[70] Mette Lundsby Jensen and Kirstine Zinck Pedersen. Utilsigtede hændelser i den kommunale plejesektor. Online: dsi.dk, 02 2010.

[71] J. A. Kalman. *Automated reasoning with Otter*. Rinton Press, 2001.

[72] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata studies*, 34:3–41, 1956.

[73] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

[74] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[75] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR'oo*, volume 1877 of *LNCS*, pages 489–503, 2000.

[76] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, May 1994.

[77] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[78] Dexter Kozen. On the coalgebraic theory of kleene algebra with tests. Technical report, March 2008. URL http://hdl.handle.net/1813/10173.

[79] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science (MSCS)*, 5(1), 1995. Conference version presented at the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL), 1993.

[80] Daniel Krob. A complete system of b-rational identities. In Mike Paterson, editor, *ICALP*, volume 443 of *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings*, pages 60–73. Springer, 1990. ISBN 3-540-52826-1.

[81] L. Lamport. The weak byzantine generals problem. *J. ACM*, 30(3):668–676, 1983. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/2402.322398.

[82] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[83] Kenny Zhuo Ming Lu and Martin Sulzmann. Rewriting regular inequalities. In *Proc. Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004*, volume 3302 of *Lecture Notes in Computer Science (LNCS)*, pages 57–73. Springer, November 2004.

[84] Karen Lyng, Thomas Hildebrandt, and Raghava Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *ProHealth '08*, 2008. URL http://www.itu.dk/people/hilde/Papers/ProHealth08.pdf.

[85] Karen Lyng, Thomas Hildebrandt, and Raghava Mukkamala. The resultmaker online consultant: From declarative workflow management in practice to ltl. In *In Proc. of 1st International Workshop on Dynamic and Declarative Business Processes (DDBP 2008)*, Munich, Germany, 2008. URL http://www.itu.dk/people/hilde/Papers/DDBP08.pdf.

[86] R. Milner. A complete inference system for a class of regular behaviours. *J. Comput. Syst. Sci.*, 28(3):439–466, 1984.

[87] Mehryar Mohri. Minimization of sequential transducers. In *Combinatorial pattern matching: 5th annual symposium, CPM 94, Asilomar, CA, USA, June 5-8, 1994: proceedings*, volume 807, page 151, 1994.

[88] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, LNCS. Springer, 2009. To appear.

[89] George C. Necula and Shree Prakash Rahul. Oracle-based checking of untrusted software. In *POPL*, pages 142–154, 2001.

[90] Uwe Nestmann. What is a "good" encoding of guarded choice? *Inf. Comput.*, 156(1-2):287–319, 2000.

[91] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. *Inf. Comput.*, 163(1):1–59, 2000.

[92] Lasse Nielsen. Regular expression compression parser. `http://www.thelas.dk/index.php/Rcp`. URL `http://www.thelas.dk/index.php/Rcp`.

[93] Lasse Nielsen. A coinductive axiomatization of xml subtyping. Graduate term project report, DIKU, University of Copenhagen, 2008.

[94] Lasse Nielsen and Fritz Henglein. Bit-coded regular expression parsing. In *LATA 2011 - 5th International Conference on. Language and Automata Theory and Applications*. Springer-Verlag, Berlin, Heidelberg, 2011.

[95] Lasse Nielsen and Fritz Henglein. Bit-coded regular expression parsing. *Submitted to International Journal of Computer Mathematics*, 2011.

[96] Lasse Nielsen, Nobuko Yoshida, and Kohei Honda. Multiparty symmetric sum types. *Arxiv preprint arXiv:1011.6436*, 2010.

[97] L. Ohno-Machado, J. H. Gennari, S. N. Murphy, N. L. Jain, S. W. Tu, D. E. Oliver, E. Pattison-Gordon, R. A. Greenes, E. H. Shortliffe, and G. Barnett. The guideline interchange format. *Journal of the American Medical Informatics Association*, 5(4):357, 1998.

[98] Luca Padovani. Fair subtyping for multi-party session types. *COORDINATION 2011*, 2011.

[99] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *MSCS*, 13(5):685–719, 2003.

[100] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[101] K. V. S. Prasad. Broadcast calculus interpreted in ccs upto bisimulation. In *Electronic Notes in Theoretical Computer Science*, volume 52, pages 83–100. Elsevier, 2001.

[102] Vaughan Pratt. Action logic and pure induction. In *Proc. Logics in AI: European Workshop JELIA*, volume 478 of *Lecture Notes in Computer Science (LNCS)*, pages 97–120. Springer, 1990.

[103] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI communications*, 15(2, 3):91–110, 2002.

[104] G. Rosu and J. Goguen. Circular coinduction. 2000.

[105] G. Rosu and D. Lucanu. Circular coinduction: A proof theoretical foundation. *Algebra and Coalgebra in Computer Science*, pages 127–144, 2009.

[106] J. Rumbaugh, R. Jacobson, and G. Booch. The unified modelling language reference manual. 1999.

[107] Jan J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In Davide Sangiorgi and Robert de Simone, editors, *CONCUR: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *CONCUR '98*, pages 194–218. Springer, 1998. ISBN 3-540-64896-8.

[108] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321312.321326.

[109] Julian Seward. Bzip. `http://www.bzip.org/`. URL `http://www.bzip.org/`.

[110] Alexandra Silva, Marcello M. Bonsangue, and Jan J. M. M. Rutten. Nondeterministic kleene coalgebras. *Logical Methods in Computer Science*, 6(3), 2010. URL `http://arxiv.org/abs/1007.3769`.

[111] Jes Søgaard, Anne Frølich, and Thomas Schiøler. Utilsigtede hændelser på danske sygehuse. Online: dsi.dk, 2001.

[112] M. E. Stickel. Resolution theorem proving. *Annual review of computer science*, 3(1):285–316, 1988.

[113] Martin Sulzmann and Kenny Zhuo Ming Lu. Xhaskell - adding regular expression types to haskell. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *IFL*, volume 5083 of *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 75–92. Springer, 2008. ISBN 978-3-540-85372-5.

[114] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.

[115] Annette ten Teije, Silvia Miksch, and Peter Lucas. *Computer-based Medical Guidelines and Protocols: A Primer and Currend Trends*. Studies in Health Technology and Informatics. IOS Press, 2008. ISBN 978-1-58603-873-1.

[116] W. M. P. Van Der Aalst and A. H. M. Ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[117] Stijn Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1133651.1133652.

[118] Margus Veanes Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proc. 3d Int'l Conf. on Software Testing, Verification and Validation*, Paris, France, April 6-10 2010. IEEE Computer Society Press.

[119] H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing workflow processes using woflan. *The Computer Journal*, 44(4):246–279, 2001. doi: http://dx.doi.org/10.1093/comjnl/44.4.246. URL http://comjnl.oxfordjournals.org/cgi/reprint/44/4/246.

[120] B. Victor and F. Moller. The mobility workbench—a tool for the π-calculus. In *Computer Aided Verification*, pages 428–440, 1994.

[121] L. Wall, T. Christiansen, and J. Orwant. *Programming perl*. O'Reilly Media, 2000.

[122] D. Wang, M. Peleg, S. W. Tu, A. A. Boxwala, O. Ogunyemi, Q. Zeng, R. A. Greenes, V. L. Patel, and E. H. Shortliffe. Design and implementation of the glif3 guideline execution engine. *Journal of biomedical informatics*, 37(5): 305–318, 2004.

[123] P. H. Welch and F. R. M. Barnes. Communicating mobile processes: introducing occam-pi. In *In 25 Years of CSP*, 2005.

[124] S. A. White. Introduction to bpmn. *IBM Cooperation*, pages 2008–029, 2004.

[125] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing series. MIT Press, feb 1993. ISBN 0-262-23169-7.