Modeling and Verification of Distributed Algorithms in Theorem Proving Environments

vorgelegt von

Dipl.-Inf.
Philipp Küfner
aus Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
-Dr.-Ing.-

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 19. September 2013

Berlin 2014
D 83

Abstract

The field of application for distributed algorithms is growing with the ongoing demand for new network applications. Wherever a global goal must be achieved by a number of peers, a distributed algorithm consisting of local computations and communication protocols must be developed. In asynchronous environments the design of a distributed algorithm must respect the fact that processes may vary in speed and that messages might be delayed for an arbitrarily long time. Moreover, it is often required that the algorithm still works in the presence of crash failures, i.e., in scenarios, where some processes stop working at some time. The complex nature of concurrent executions in distributed systems often hinder the application of common methods for modeling and verification. Although many different formal approaches for modeling fault tolerant distributed algorithms, like e.g. I/O automata and process calculi have been proposed, many works in the field of distributed systems use informal pseudo-code representations to introduce new algorithms. On most cases this leads to informal and crude arguments for correctness, which usually become incomprehensible if all race conditions must be taken into account.

This work proposes new formal strategies to model and verify distributed algorithms, whereas we focus on two main goals. Firstly, we introduce a means to satisfy a designer's demands by using methods that are easy to understand and to work with and which are very similar to common applied methods as e.g. TLA$^+$ and Abstract State Machines. Secondly, our strategies are suitable for the formal use in a theorem proving environment. This enables mechanical verification of our algorithms and therefore, we are able to produce machine-checked proofs for correctness.

Our model adopts abstractions from the work of Fuzzati, which examines two similar distributed algorithms in a logical framework. We lifted these abstractions to a more general level to make them suitable and reusable for as many distributed algorithms as possible. Furthermore, we present a new method to place local computation steps in the respective global context. The advantage of our method is that, on the on hand, like with pseudo code, computations can be described as local steps, but, on the other hand, it is possible to formally reason over global system states. Our model for communication mechanisms comprises modules for different kinds of message passing, broadcasts, and shared memory access. We are convinced that our model can easily be extended to support almost all kinds of communication infrastructures that can formally be described.

We show how properties that have to be verified can be expressed in terms of our model to enable formal reasoning. Furthermore, we present appropriate strategies to verify different classes of properties.

For the theorem proving environment Isabelle/HOL we provide a library which incorporates all introduced abstractions and can, therefore, be reused and extended for further applications, new algorithms, and further verification projects.

Finally, we present case studies for the verification of different distributed algorithms to demonstrate the applicability of our strategies. Four known algorithms are verified in our model using our framework for Isabelle/HOL. The algorithms vary in complexity, the used communication infrastructure, the failure model, and other constraints, so that many different modeling and verification aspects are exemplified.

Zusammenfassung

Mit dem wachsenden Markt für Netzwerkanwendungen gewinnt das Anwendungsgebiet von verteilten Algorithmen immer mehr an Bedeutung. Sobald ein globales Ziel durch ein koordiniertes Zusammenwirken mehrerer Prozesse erreicht werden muss, müssen lokale Berechnungen auf die einzelnen Teilnehmer verteilt und Kommunikationsprotokolle festgelegt werden. Dies erfordert die Entwicklung eines verteilten Algorithmus. In asynchronen Umgebungen müssen verteilte Algorithmen so konstruiert sein, dass sie das globale Ziel trotz unterschiedlicher Laufzeiten der Teilnehmer und verzögert ausgelieferter Nachrichten erreichen. Oft ist es eine zusätzliche Anforderung, dass verteilte Algorithmen auch in Hinblick auf den Ausfall einzelner Teilnehmer fehlertolerant sind. Die zusätzliche Komplexität, die durch die nebenläufige Ausführung der Prozesse in verteilten Systemen entsteht, macht den Einsatz konventioneller Methoden zur Modellierung und Verifikation der verteilten Algorithmen häufig unmöglich. Obwohl es viele verschiedene formale Ansätze, wie z.B. I/O Automaten und Prozesskalküle, zur Modellierung fehlertoleranter verteilter Algorithmen gibt, ist das verbreitete Mittel zur Darstellung neuer Algorithmen Pseudocode. Diese wenig formale Modellierung ermöglicht keine formale Verifikation und daher ist die Beweisführung zur Korrektheit der Algorithmen meist eher eine unpräzise Argumentation, die durch die Vielzahl der möglichen Ausführungsszenarien von verteilten Algorithmen häufig nicht nachvollziehbar ist.

Inhalt dieser Arbeit ist die Erarbeitung neuer formaler Strategien zur Modellierung und Verifikation verteilter Algorithmen unter besonderer Berücksichtigung zweier Ziele: Zum einen neue Methoden zur Verfügung zu stellen, die den Ansprüchen eines Algorithmendesigners genügen, d.h. möglichst einfach zu verstehen und aufwandsarm anzuwenden sind. Auf der anderen Seite sollen diese Methoden formal genug sein, um die Anwendung maschineller Verifikation in Form von Theorembeweisern und damit eine mechanische Beweisführung für die Korrektheit zu ermöglichen. Um das erste Ziel zu erreichen verwenden wir Methoden, die bisherigen praxisnahen Modellierungsmethoden, wie z.B. Leslie Lamports Spezifikationssprache/-logik TLA$^+$ oder Gurevichs Abstract State Machines sehr ähnlich sind.

Zur formalen Modellierung von Algorithmen verwenden wir viele Abstraktionen, die der Arbeit von Fuzzatientnommen sind. Fuzzatis Arbeit untersucht zwei sehr ähnliche Algorithmen auf der Basis einer formalen prädikatenlogischen Modellierung. Diese Arbeit greift diese Abstraktionen auf und verallgemeinert sie so, dass sie zur passend und wiederverwendbar für eine Vielzahl von Algorithmen sind. Zusätzlich stellt diese Arbeit eine neue Methode vor, wie lokal beschriebene Berechnungsschritte auf einfache Weise in einen globalen Ausführungskontext übertragen werden können. Der Vorteil dieser Methode ist, dass Schritte, die lokal vollzogen werden, lokal beschrieben werden, aber dennoch in der Beweisführung für ein globales Ziel im globalen Kontext eingebettet werden können. Entsprechend ist die Modellierung aufwandsärmer, da lokale Berechnungsschritte, ähnlich wie im Pseudocode, ohne die Beschreibung des weiteren globalen Rahmens modelliert werden können. Gleichzeitig ermöglicht die formal korrekte Einbettung in den globalen Kontext formale Beweise zum Nachweis globaler Systemeigenschaften. Unser Modell umfasst dabei mehrere verschiedene Kommunikationsmechanismen wie Punkt-zu-Punkt Nachrichten, Broadcasts und Kommunikation über gemeinsam genutzte Speichersegmente. Erweiterungen dieser Möglichkeiten um weitere mögliche Kommunikationsinfrastrukturen zu unterstützen können einfach in das Modell integriert werden.

Neben der Modellierung ist ein weiterer Schwerpunkt dieser Arbeit die Analyse verschiedener Strategien zum formalen Nachweis von Systemeigenschaften. Dazu müssen diese Eigenschaften zunächst in unserem Modell beschrieben werden. Diese Arbeit stellt Methoden zur Formulierung und Verifikation der beiden wesentlichen Klassen von Eigenschaften (Safety- und Livenesseigenschaften) vor. Für den Theorembeweiser Isabelle/HOL stellen wir ausserdem eine Bibliothek zur Verfügung, die die vorgestellten Abstraktionen und ebenso die Strategien zur Beweisführung in Form von Definitionen und Theoremen kapselt und dadurch wiederverwendbar macht.

Der letzte Teil der Arbeit erbringt dann anhand mehrerer Fallstudien den Nachweis der Anwendbarkeit der vorgestellten Strategien. Vier bekannte verteilte Algorithmen werden in unserem Framework in Isabelle/HOL untersucht und verifiziert. Die vier Algorithmen variieren dabei in mehreren Aspekten, wie Komplexität, Kommunikationsinfrastruktur, Fehlermodell und Annahmen über die Synchronität des Systems und veranschaulicht daher viele wesentliche Modellierungs- und Verifikationsaspekte.

Contents

Acknowledgements

Firstly, I like to thank Prof. Dr. Ing. Uwe Nestmann for initiating this thesis, providing scientific support and giving me the opportunity to finish this work under excellent conditions. I am very grateful for the time when I have been a part of his group ‘Models and Theory of Distributed Systems’ (MTV). I would like to express my very great appreciation to my co-supervisor PhD habil. Florian Kammüller for many useful advices and discussions, which helped me to broaden my horizon and recognize that many details of this work have counterparts in other theoretical and practical applications.
Furthermore I wish to acknowledge the help provided by Sven Schneider, who always had the patience to discuss all the technical details and furthermore gave me many useful hints especially on Chapter 2 after the first version was finished.
Special thanks should be given to Dr. Stephan Merz for his patient introduction to Isabelle/HOL and for his advices in modeling distributed algorithms.
I would also like to extend my thanks to the students who participated in our Isabelle/HOL projects: Christina Rickmann, Matthias Rost, Étienne Coulouma, Benjamin Bisping and Liang Liang Ji.
I would also like to thank the rest of the group at MTV for their discussions during our research colloquiums.
English language assistance provided by Deborah Doughton was greatly appreciated.
Finally, I wish to thank Vivien Hucke for her great support and encouragements throughout this work and I would like to thank my parents Monika Robl-Küfner and Joachim Küfner and my sister Anna Küfner for their ongoing support in every aspect of my life.

1. Introduction

1.1. Overview

The world wide web and other global and local networks have become an essential part of modern life. The example of cloud computing shows that there is a demand for distributed systems not only in research but also commercially. Distributed systems, now a part of standard applications, take on even more importance as mobile computing rapidly increases. With the growing demand for interacting and communicating devices, the need for appropriate algorithms increases. The efficiency of these algorithms might be desirable but their correctness is a necessary condition.

Many models and methods exist to analyse, test and verify algorithms and programs, but only a few of them are suited for distributed computing. The most common and well-known method for reasoning over sequential algorithm has been proposed by Hoare [Hoa69], and others have followed. However, characteristics of distributed algorithms hinder the use of standard verification methods in environments where more than one process is involved. Attiya and Welch [AW04] mention three fundamental difficulties that arise in distributed situations:

- asynchrony
- limited local knowledge
- failures

Attiya and Welch [AW04] identify the model of computation, as the main difference compared to the sequential case. For a non-randomised sequential algorithm, executions are fully determined by the input values. Hence, there is only one single universally accepted model of computation for the sequential case. In most distributed cases there are many possible executions (this will further be explained later).

Therefore, the increasing importance of concurrent algorithms for practical applications causes a need for adequate theories to model these algorithms and to demonstrate their correctness, i.e., to verify that the algorithms do fulfil requirements.

Analysis of algorithms can be divided into two categories:

- dynamic analysis, i.e. inspecting the executions for errors (e.g. testing and model checking)
- static analysis, i.e. inspecting the code/algorithm without doing an execution (e.g. syntax checking and formal verification)

If an algorithm can produce an infinite amount of different executions, the inspection of executions can only prove correctness for a subset of all possible cases. In contrast, static analysis includes verification methods that allow assertions for every execution.

Using theorem provers can be regarded as the most rigourous method for formal verification. Chou claims in [Cho95] that 'the only practical way to verify the correctness of distributed algorithms with a high degree of confidence is to construct machine-checked, formal correctness proofs'. This work analyses ways to model and verify distributed algorithms and presents an appropriate model that respects, on the one hand the belongings of the designer and, on the other hand, it remains formal enough to be used in a theorem proving environment.

1.2. Scope of Work

A quote from Leslie Lamport states that 'A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable'. In this work we restrict the algorithms to be analyzed to systems with a fixed number of involved processes. We adapt the definition from [AW04] to:

Definition 1.2.0.1 (Distributed System)
A distributed system $\mathfrak{D}_N$ is a collection of N individual computing devices (processes) that can communicate with each other.

A more formal definition of a distributed algorithm will be given in Chapter 2. Modeling-approaches for distributed algorithms differ in many aspects. Some describe local behaviours (e.g. pseudo code descriptions) while others describe global behaviours in terms of state machines that can interact with each other by using shared actions (e.g. process calculi as [Mil82] and I/O automata [LT87]). Both, the level of abstraction a model uses and the formal semantics which underly the model's abstractions, are important aspects in considering the verification of distributed systems. While semantics for process calculi and I/O automata are well defined, there is no formal semantics for pseudo code. Nevertheless, from a designer's point of view, pseudo code is often the model of choice since it is easy to understand and closer to common programming languages. It seems as process calculi and I/O automata are too far from the natural way to describe how a program has to behave. For that reason, many algorithms and also distributed algorithms are given as pseudo code fragments and the arguments for correctness refer to these pieces of informal code. Naturally, since there is no interpreter, no compiler, and no formal semantics for pseudo code these arguments always depend on the interpretation of the code. Hence, more formal methods are needed for a formal verification of algorithms.

The intent of this work is to develop strategies that enable us to transfer a given piece of code to a model that is formal enough to mechanically check proofs for correctness. Moreover, this work provides a framework for the theorem prover Isabelle consisting of several libraries which support the design of new algorithms and provide theorems for the meta theory of the model. Based on case studies in which several distributed algorithms are verified we developed the libraries. We finally succeeded in verifying those algorithms with a theorem proving environment.

1.3. Modeling and Verification

At first glance, modeling and verification might be regarded as two independent tasks during the development process of a distributed system. But in fact verification depends on the former modeling process because verification can only be done in terms of already defined components. If we want to reason about system components, we need to first specify all components and how these modules behave during the execution of the system. Therefore, we consider the model of a system as a formal specification of the algorithm and its environment.

The most concrete verification scenario would be a setting in which an algorithm is implemented in some programming language. The subsequent verification task would be to prove that the implementation meets the exact demands of the customer or user of the system. In this case, a model would be the implementation, and verification would imply simply entering every possible input value and checking that the system returns the right output. As long as inputs are finite and the system consists only of one sequential process, this method seems reasonable, and, if there is enough time available, this method is also applicable. But for distributed systems in which two or more processes run in parallel, this approach turns out to be infeasible. The following example demonstrates this thesis and, at the same time, motivates our approach for formal reasoning.

1.3.1. Motivation

For example, assume N processes have to execute the same algorithm consisting of ten sequential operations $\mathcal{A}_0, \ldots, \mathcal{A}_9$. For a distributed system we would expect the processes to work in parallel or at least concurrently. For most cases, (theoretical and practical), we are not able to say whether processes will need exactly the same amount of time to execute a single command $\mathcal{A}_i$. Moreover, commonly, there is no assumption on synchronization between the processes. Hence, we do not know which process will finish the execution of $\mathcal{A}_0$ first or when the last process will finish the program by executing $\mathcal{A}_9$. Maybe there is one process that has finished the program before all other processes have even started with the first operation $\mathcal{A}_0$. Usually we do not know whether it will always be the same process that finishes first. In some systems these questions do not matter (for example, if processes work autonomously), but for most distributed systems these questions are very important due to interdependencies between operations of different processes. Assume, for example, that processes communicate by sending messages to each other and an action $\mathcal{A}_i$ executed by a process p_j at time t depends on the messages p_j has previously received. Naturally, the messages p_j has received until t will depend on the actions the other processes executed before t. Hence, the first observation we make is: to prove correctness, we need a model of computation. If we want to prove that our system always satisfies the requirements, we have to consider all possible cases of computations that might happen, and then we need to show that requirements are met in every scenario. Hence, at first we would have to analyse which scenarios are possible within our model. Let us assume the processes are threads in Java and that they run on a single processor machine such that there can only be one action at a time. Let $\mathcal{A}_0, \ldots, \mathcal{A}_9$ be the instructions the threads execute. Of course, we would not be able to make any assumptions about how the Java Virtual Machine will schedule the different threads. Therefore, it would be required to regard every possible schedule. Even if we only have two threads, we would have to consider up to $\frac{20!}{10! * 10!} = 184756$ scheduling options (which correspond to 184756 traces). It is obvious that, to prove correctness, we need the right strategy to reduce these cases in our proofs. As long as we deal with a (preferably small) finite number of cases we still can prove correctness by doing model checking (test every trace). However, if every process executes those instructions in an infinite loop we obtain an infinite number of cases. Algorithms that run infinite loops are very common for distributed algorithms, and, therefore, the model checking approach works only for a very limited number of algorithms. Hence, to verify our algorithms mechanically we do not make use of model checking techniques but analyse how theorem proving can be applied to distributed algorithms.

1.3.2. Our Strategies and Organization of This Work

As mentioned previously, to verify an algorithm it is important to have a suitable model of computation. This model must be able to represent every possible execution of the algorithm. Therefore, we analyse different aspects of modeling in Chapter 2, introducing our formal model of computation in Section 2.3. Furthermore, for the process of doing the proofs it is, necessary to specify the requirements for correctness in terms of the model. Therefore, Section 2.4 describes how we can express the desired properties by the means introduced in Section 2.3. As we have already demonstrated, an important aspect for a successful verification approach is the right strategy to do the proofs. For this purpose, in Chapter 4 we explain different methods, techniques and strategies we have applied for the verification of some sample algorithms. The respective case studies are then further explained in Chapter 5, which consists of the verification of four algorithms with different complexity. Finally, we present our conclusions in Chapter 6.

For all results in this work we provide formal proofs written down as proof scripts for Isabelle/HOL. Therefore, this work provides references to the formal proofs for all lemmas, propositions and theorems as footnotes. The following example shows how we reference to a provided Isabelle theory file `Example.thy`, containing a verified lemma with name `MyLemma`: ↪ Isabelle[1]. All formal proofs are provided in the respective files on the enclosed CD-rom. Selectively, we also provide informal proof ideas to give the reader an idea how the proof was done.

1.3.3. Summary of Contributions

This work presents a new asynchronous model to represent Distributed Algorithms in theorem proving environments (see Chapter 2). Beside the feature that it is formal enough to work with theorem provers, our model is closely related to well-known modeling approaches such as TLA and [abstract] state machines such that our models can be transfered into TLA or state machine representations and vice versa. The model also comprises definitions for different communication mechanisms such as shared memory (Regular Registers) and message passing. It has general application for a large group of algorithms. To the best of our knowledge, our model of Regular Registers is the first state-based approach that is formal enough to be used in a theorem proving environment. We also describe how different failure models can be implemented in our model. Finally, we consider how to avoid errors while using our model and provide constraints to give a guarantee of consistency for modeled algorithms deliberating our notion of distribution.

The right strategy to verify an algorithm depends on the character of its properties that have to be shown. We provide means for proving both, safety and liveness properties (see Chapter 3).

[1]Isabelle/HOL theory: `MyLemma`, Lemma(s): `Example.thy`

To support modeling and verifying new algorithms in a theorem proving environment, we provide a library for the theorem prover Isabelle/HOL. This library implements the theory described in Chapters 2 and 3 by providing types, datastructures, definitions and theorems to comfortably model and verify a given distributed algorithm using different communication infrastructures and failure models.

Finally, we checked the applicability of our concepts to different case studies (see Chapter 5). In these case studies we verified algorithms of different complexities by different means for communication and different failure models.

1.4. Technical Preliminaries

1.4.1. Problems in Distributed Systems

An important comprehensive work on distributed algorithms is [Lyn96]. It presents solutions to well-known distributed problems as, e.g.:

Leader Election Eventually exactly one process should output the decision it has become the leader [Lyn96].

Mutual Exclusion As N cyclic processes enter and exit critical sections, the processes have to be programmed in such a way that at any one moment only one of theses processes is in its critical section [Dij65]

Distributed Consensus The basic setting is a network of N processes that have to agree on exactly one value. Therefore, an algorithm that solves Consensus is supposed to exhibit the follwing properties ([Lyn96], [CBS07]):

- Agreement: No two processes decide on different values
- Validity: If a process p_i decides a value v then there has to be a process p_j such that v has been input for p_j.
- Termination: All nonfaulty processes eventually decide.
- Irrevocability: Once a process decides on a value, it remains decided on that value.

Distributed Consensus is the problem this work focuses on. One main criterion affecting the solvability of a problem is the degree of synchronicity of the system. In completely asynchronous systems there is no guarantee for the message delay and no timing assumptions for local command executions. This can be an extreme handicap, especially while reasoning about liveness properties of a system.

In fact the work of Fischer, Lynch and Paterson [FLP85] shows that Distributed Consensus (or for short Consensus) is not solvable in asynchronous environments with crash failures. Hence, there are many studies (e.g. [CHT96], [CBM09]) proposing additional assumptions which make Consensus solvable in asynchronous environments.

Consensus belongs to the class of Agreement problems. Agreement problems are fundamental to many distributed algorithms, because the task of disseminating and agreeing on a common knowledge is a requirement for almost every application that uses distributed services. Descriptions of other commonly known Agreement problems (as for example Atomic Commitment and total order broadcast) can be found in [GR06].

1.4.2. Theorem Provers

Theorem provers are computer programs that are able to do formal proofs either automatically or in interaction with the user. Theorem provers, like the one considered in this work, can be regarded as verification tools that can be used for a multitude of puposes. A theorem prover enables the user to give a machine-checked proof that a certain statement (the conjecture) is a logical consequence of a set of statements (axioms and hypotheses).

The application of a theorem prover requires the user to expose axioms, hypotheses, and conjectures in a very detailed way. This might be regarded as a handicap because it requires some effort to model all logical details. But this effort is wellworth its while if an error-prone design process can be turned into an almost faultless development. An effective example for this is the industrial application of theorem provers in hardware verification.

Theorem provers can be differentiated by the degree the user is involved in the proof process, i.e. there are 'Automated Theorem Provers' like Otter[Mcc] and SPASS[WDF$^+$09] and 'Interactive Theorem Provers' that require the user to guide the software through the proofs by naming the inference rules for the single steps of the proof or by writing and applying tactics to deduce proof goals from the axioms. Some well-known examples for interactive provers are HOL[Hut94] and Coq[FHB$^+$97].

The input language for the theorem prover restrains the opportunities for the specification of the model for the algorithm to inspect. Hence, another important criterion for the selection of a theorem prover is the input language for the formulae. In most cases, this is first order logic (e.g. Otter, SPASS) or higher order logic (HOL, Coq). While the meta-language for HOL is ML, terms are drawn from typed λ-calculus and classical predicate logic.

Isabelle/HOL

For the mechanical verfication of algorithms we use the interactive theorem prover Isabelle in this work. Nipkow et. al [NPW02] describe Isabelle as a 'generic system for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which abbreviates Higher-Order Logic'. Such a specialization of the generic theorem prover is called the *object-logic*. Also, Isabelle's logical framework, the *meta-logic* of Isabelle, uses higher-order logic based on the typed λ-calculus [Pau89]. An overview of the object logic HOL can be found in [NPW]. Beside HOL, Isabelle supports several other logics (e.g. ZF a Zermelo-Fraenkel set theory, which is build on top of FOL (First Order Logic) [PNW03]). We have chosen to use the HOL specialization of Isabelle because it meets our requirements and, according to Paulson, is 'the best developed Isabelle object-logic, including an extensive library of (concrete) mathematics and various packages for advanced definitional concepts' [Pau13]. Isabelle, itself, is implemented in the general-purpose functional programming language ML (see [MTM97]).

Every proof of a theorem (respectively lemma, proposition, corollary. . .) in Isabelle is given as a proof script. A proof script can be seen as a list of commands which tell the theorem prover how to obtain the proof. These commands in Isabelle can be e.g. references to proof rules (as e.g. modus ponens etc.), references to lemmas and theorems that have been proven earlier, or references to Isabelle methods, like `auto`, `simp` or `induct`. One step in a formal proof corresponds to one application of a proof rule (respectively theorem or lemma). Sophisticated methods like `auto` and `simp` try to automatically determine which proof rules to use next. Thereby `simp` uses only simplification rules. The set of simplification rules is adaptable. Heuristics like `auto` can be used to make Isabelle do the proof work for small proof steps. Figure 1.1 shows how the simple theorem

$$2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq n \,\}} i \right) = (n+1) \cdot n \tag{1.1}$$

can be proved by induction and the extensive use of `auto` in Isabelle/HOL. First the name of the theorem is declared as `SumNat`. Then the keyword `shows` introduces the assertion we want to prove, which is here Equation 1.1. We tell Isabelle that `i` is of type `nat` (which corresponds to the natural numbers in Isabelle) by writing `i::nat`. The function `Suc` denotes the successor function (e.g. we have `(Suc n) = (n + 1)`). Note that Isabelle allows to omit the brackets for function application and hence we have `(Suc 1) = (Suc(1))`. After the keyword `proof` we write the method we want to apply to our current proof goal. At the beginning, our proof goal is the conjecture of the theorem and hence Equation 1.1 and we want to prove this goal by induction over `n`. Hence we apply method `induct` with parameter `n`. This generates two new proof goals:

- the base case: $2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq 0 \,\}} i \right) = (0+1) \cdot 0$
- the step: For an arbitrary fixed `n` we have to show that the induction hypothesis

$$2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq n \,\}} i \right) = (n+1) \cdot n \tag{IH}$$

  implies the assertion

$$2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq (n+1) \,\}} i \right) = ((n+1)+1) \cdot (n+1).$$

The base case is shown by first proving the following equation

$$\{\, i \in \mathbb{N} \;\mid\; i \leq 0 \,\} = \{\, 0 \,\}$$

which is derived by using method `auto`. This result is used to show

$$2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq 0 \,\}} i \right) = 0,$$

which finally implies the base case.

For the inductional step we fix a variable *n* using the keyword *fix* and assume (by the respective keyword *assume*) the induction hypothesis. Again method *auto* is used to derive

$$\{\, i \in \mathbb{N} \;\mid\; i \leq (n+1) \,\} = \{\, i \in \mathbb{N} \;\mid\; i \leq (n) \,\} \cup \{\, n+1 \,\}\,.$$

This yields

$$\sum_{\{\, i \in \mathbb{N} \mid i \leq (n+1) \,\}} i = \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq n \,\}} i \right) + n + 1$$

With the induction hypothesis we can infer

$$2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq (n+1) \,\}} i \right) = ((n+1) \cdot n) + 2 \cdot (n+1)$$

Hence we have

$$2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq (n+1) \,\}} i \right) = (n+2) \cdot (n+1)\,,$$

which finally yields the step conjecture of the induction:

$$2 \cdot \left(\sum_{\{\, i \in \mathbb{N} \mid i \leq (n+1) \,\}} i \right) = ((n+1)+1) \cdot (n+1)\,.$$

Every described step corresponds to one line in the Isabelle proof script in Figure 1.1. For our proofs we use the language *Isar*, which is recommended by Nipkow [Nip13] for larger proofs since it is structured and not linear, and proofs become more readable without running Isabelle. Compared to our informal description of the proof, we see that the description is very similar to the proof script.

The work with Isabelle is structured in theories, which are named collections of types, functions and theorems [NPW02]. Therefore, if we give references to our Isabelle proof scripts, we will name the theory for the respective theorem or definition. A theory with name `x` must be located in a file `x.thy`.

In the following we explain some of Isabelle's keywords and constructs that are important for this work. We will not explain the technical aspects of the proofs in this work but only describe how we express our models in Isabelle presenting the most important theorems and proof ideas. Therefore we focus on definitions and types in the following description and refer the reader to [NPW02] and [Nip13] for more information on Isabelle's proof languages.

A theory starts with the keyword **theory** followed by the name of the theory. After the name it is possible to import other theories by the keyword **imports** followed by the name of the imported theory. Then the keywords **begin** and **end** enclose the body of the theory.

A non recursive definition in Isabelle is introduced by the keyword **definition**. A definition of an object T at first requires defining the type of T. For example, a function $f : \mathbb{N} \rightarrow \mathbf{bool}$ where $f(n)$ is true if and only if $n > 5$ can be defined as follows:

```
theorem SumNat: shows "2*(∑i ∈ {i::nat. i ≤ n}.i) = (Suc n)*n"
proof (induct n)
  have "{(i::nat). i ≤ 0} = {0}" by auto
  hence "∑{(i::nat). i ≤ 0} = 0" by (auto)
  thus "2 * ∑{(i::nat). i ≤ 0} = (Suc 0) * 0" by auto
next
  fix n
  assume IH: "2 * (∑ i ∈ {i. (i::nat) ≤ n}.i) = (Suc n) * n"
  have "{i. (i::nat) ≤ Suc n} = {i. i ≤ n} ∪ {Suc n}" by auto
  hence "(∑i ∈ {i. (i::nat) ≤ Suc n}.i) = (∑i ∈ {i. (i::nat) ≤ n}.i) + Suc n"
    by auto
  hence "2 * (∑i ∈ {i. (i::nat) ≤ Suc n}.i) = 2 * (∑i ∈ {i. (i::nat) ≤ n}.i) + 2 * (Suc n)"
    by auto
  with IH have "2 * (∑i ∈ {i. (i::nat) ≤ Suc n}.i) = (Suc n) * n + 2 * (Suc n)"
    by auto
  hence "2 * (∑i ∈ {i. (i::nat) ≤ Suc n}.i) = (n+2) * (Suc n)" by auto
  thus "2 * (∑i ∈ {i. i ≤ Suc n}.i) = Suc (Suc n) * Suc n" by auto
qed
```

Figure 1.1.: Application of method `auto` in theorem SumNat

```
definition f:: "nat ⇒ bool" where
  "f n ≡ (n > 5)"
```

Commonly, currying (cf. [Cur58]) is used for the definition of functions that require more than one argument. Hence a function $g : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ with $g(n, m) \triangleq n + m$ is defined in Isabelle as follows:

```
definition g:: "nat ⇒ nat ⇒ nat" where
  "g n m ≡ n + m"
```

Functions in Isabelle are always total functions. To define a function that is undefined for some domain values the `option` datatype can be used. A variable of type `'a option` is either `None` or `Some v` where `v` is a value with type `'a`. Hence the partial function $h : \mathbb{N} \rightharpoonup \mathbb{N}$ with

$$h(n) \triangleq \begin{cases} n-1 & \text{, if } n > 0 \\ \bot & \text{, otherwise} \end{cases}$$

can be defined in Isabelle as follows:

```
definition h:: "nat ⇒ nat option" where
  "h n = (if (n > 0) then (Some (n - 1)) else None)"
```

Isabelle provides an `if ...then ...else` construct to implement the case distinction. Note that it is not possible to treat a return value of `h` like a natural number because the type of `h n` is `nat option` and not `nat`. To work with the value `v` in `Some v` there is a function `the` such that `the(Some v) = v`. Hence `(h n) + 5` yields a type error but `(the (h n)) + 5` is an expression of type `nat`. Note that `(the None)` is not defined, i.e., `the(h 0)` is of type `nat`, but the value of `the(h 0)` is not defined and hence unknown.

The keyword **datatype** permits the definition of new datatypes. A standard example is the predefined definition of lists in Isabelle. We can define a list of elements of type `'a` by using the following datatype:

datatype 'a list = Nil | Cons 'a "'a list"

Note that `'a` is a type variable and hence the definition allows defining lists for every type, such that e.g. `nat list` is a datatype for a list of natural numbers and `bool list` is a list of boolean values. Lists are inductively defined: `Nil` respectively `[]` represents the empty list. The constructor `Cons` takes two arguments: the head of the list (with type `'a`) and the tail of the list (with type `'a list`). Hence `Cons 1 ((Cons 2) Nil)` is the list `[1,2]`. For a list `xs` Isabelle offers functions `hd` and `tl` such that `hd xs` returns the head and `tl xs` returns the tail of the list `xs`. More information on lists and datatypes are also given in [NPW02] and [Nip13].

The keyword **type_synonym** allows introducing synonyms for existing types. Therefore

type_synonym T = nat

introduces the type synonym `T` for the type `nat` such that wherever the type `nat` is used we can also use type `T`.

The other constructs we have used are more or less self-explanatory and described when used. A detailed description of how Isabelle can be used is given in [NPW02] and some introduction can be found in [Nip13].

1.4.3. Notation for Logic and Functions

In general our notation to denote our logical formulae and models is based on the syntax and semantics of higher-order formulae in the theorem-prover Isabelle/HOL (see Section 1.4.2). We have amended only some minor details. This section explains the most important elements of the notation we use.

For definitions we use the symbol $\triangleq$. Hence

$$P \triangleq Q$$

means P is defined as Q. $\triangleq$ has lowest priority and, therefore, $P \triangleq Q$ abbreviates $(P) \triangleq (Q)$.

We define the set of boolean values as

$$\mathbf{bool} \triangleq \{\ \mathrm{True},\ \mathrm{False}\ \}$$

True represents a tautology and False the absurdity (cf. [NPW]). A function with domain A and co-domain B has type $A \to B$. To denote such a function we write $f : A \to B$. As explained in the previous section, due to currying a function $f : (A_1 \times A_2 \times \ldots \times A_n) \to B$ is represented by $f' : A_1 \to (A_2 \to (\ldots \to (A_n \to B)))$ in Isabelle. We will use the common notation $((A_1 \times A_2) \to B)$ for our logical description, but use the curried version in Isabelle.

We use the common logical connectives $\wedge, \vee$ for logical 'and' and 'or' and $\neg$ for negation. We will use the symbol $\to$ to represent steps of an algorithm in the later chapters and, therefore, in contrast to Isabelle's object logic HOL, we use the symbol $\Rightarrow$ for implication to avoid ambiguities. Furthermore we use $P \leftrightarrow Q$ as an abbreviation for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

We use the common binders $\forall$ and $\exists$ for universal and existential quantification. Precedences are assumed as defined for Isabelle/HOL. Hence, priorities are transitively ordered as follows: $\neg$ has the highest priority, $\wedge$ has higher priority than $\vee$, $\vee$ has higher priority than $\Rightarrow$ and $\Rightarrow$ has higher priority than $\forall$ and $\exists$. Therefore $\exists x \,.\, P(x) \vee Q(x)$ is equivalent to $\exists x \,.\, (P(x) \vee Q(x))$ (cf. [NPW]). Priorities for equality etc. $(=, \leq, <, \geq, >)$ are higher than for the logical connectives. Logical connectives are right-associative but $=, \leq, <, \geq, >$ are left-associative.

We use the notation $\forall x \in A \,.\, P(x)$ to abbreviate $\forall x \,.\, x \in A \Rightarrow P(x)$ and we write $\exists x \in A \,.\, P(x)$ to denote $\exists x \,.\, x \in A \wedge P(x)$. We also allow binders to bind more than one variable, e.g., we write $\forall x, y \in A \,.\, P(x, y)$ to abbreviate $\forall x \in A \,.\, \forall y \in A \,.\, P(x, y)$.

We use the Hilbert description operator ϵ to deterministically choose a value that satisfies a given condition. Hence $\epsilon x.\ P(x)$ is some value x satisfying P. If there is no such value, then the value of $\epsilon x.\ P(x)$ is undefined. Hence we can only deduce $P(\epsilon x.\ P(x))$ if $\exists x \,.\, P(x)$.

For a more detailed introduction to syntax and semantics of HOL the reader is refered to [NPW].

1.5. Related Work

The work of Chou in [Cho94] describes a model for mechanical verification of distributed algorithms in terms of Fair Transition Systems (FTL). An FTL is defined by a quadrupel (I, N, X, A) where I is a predicate which describes the conditions for the initial states, N describes the possible transitions and X and A describe the fairness assumptions. As we will see in Section 2.3, their model is very similar to this work, but we have extended the model further by attuning it to the following aspects:

- We distinguish between global events and local actions (cf. Section 2.3.1).
- We introduce means to guarantee certain constrains for consistency of our model (cf. Section 2.3.2).
- We introduce strategies to more conveniently describe local behaviour (cf. Section 2.3.3).
- We introduce reusable modules supporting different kinds of communication mechanisms (cf. Section 2.3.5).
- We explain how different kinds of properties can be defined and verified in our model (cf. Section 2.4 and Chapter 3).

[Cho94] exemplifies their method by the verification of a simple distributed algorithm called Dsum. Their work concludes with the remark that they want to verify more complex distributed algorithms in their future work and that they want to automate as many parts of their proofs as possible. Both goals are main objectives of our work. Moreover, in this thesis we examine algorithms that are much more complex (cf. Chapter 5) than the considered Dsum algorithm.

The work of Toh Ne Win [Win03] introduces techniques to make theorem-proving for distributed algorithms more effecient. By generating and analyzing test executions Win tries to automatically detect invariants. His work also presents a case study consisting of three algorithms exemplifying how methods can be applied. Models and proof methods in [Win03] are built upon a formalization of I/O automata and use Isabelle/HOL and Larch Prover as theorem proving environments. Although the goals of [Win03] seem very similar to ours, the approach differs in many aspects:

- [Win03] does not analyse which model might work well for the formalization but simply uses a standard model (I/O automata)
- [Win03] uses some applications outside the theorem proving environment (namely the Daikon invariant detector) to establish potential proof goals and lemmas in the theorem provers. In contrast, we focus on methods to make proving within the theorem proving environment more efficient.
- Win's [Win03] focus is on safety properties, whereas we consider all kinds of properties that matter for fault tolerant computing.

Our work is mainly inspired by the work of [JM05], which verifies the Disk Paxos Algorithm given by Lamport and Gafni in [GL00]. The model that is used in [JM05] is some formalization of Lamport's specification language TLA (cf. [Lam02]), which combines temporal logic with a notion of actions. We apply the TLA-action format for our formalizations in Isabelle, but we extend the model of Jaskelioff and Merz adding the important notion of runs enabling us to prove liveness properties.

In a joint work with Charron-Bost, Merz also proposed using the *Heard-Of Model* for verification purposes within the theorem prover Isabelle/HOL [CBM09]. The Heard-Of Model is introduced by Charron-Bost and Schiper in [CBS07] as a new computational model for fault-tolerant distributed computing. The model is based on the notion of transmission faults and assumes that algorithms evolve in rounds. Although used to model asynchronous systems, the model regards rounds as *communication-closed layers*. Therefore, proofs seem to be much simpler since an induction on round numbers is a natural proof method for this model. Although the algorithms in our case study use some notion of round numbers, we do not want to restrict the algorithms to round-based algorithms by paradigm. Therefore we chose a more general model of computation. For our Isabelle formalizations we applied some techniques from [CBM09] to make our proofs reusable. This will be further explained in Chapter 4.

Finally, many ideas of our model are already developed in [Fuz08] and [FMN07] (we will give detailed references at the respective parts of this work). Whereas these works give modeling advices by presenting two elaborate examples, our approach provides general theory for modeling distributed algorithms. Moreover [Fuz08] aims at giving a more formal model to enable more detailed reasoning but, nevertheless, still in form of paper proofs. This work also provides concepts and libraries to model and verify distributed algorithms within the theorem proving environment Isabelle/HOL.

Further references to related work will be given throughout the work at the relevant passages.

1.6. Own Work

Parts of this thesis have been published in

> [KNR12] Philipp Küfner, Uwe Nestmann, and Christina Rickmann: **Formal verification of distributed algorithms - from pseudo code to checked proofs.** In Jos C.M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science.* Springer 2012

I, Philipp Küfner, am principal author of [KNR12]. The article summarizes main ideas of my thesis, but many concepts and details have been worked out after publication of [KNR12] or were not mentioned due to space limitations. Hence the model presented in this thesis shows many enhancements compared to the basic ideas described in [KNR12]. For example, the shared memory abstraction presented in this work has not been part of the work in [KNR12]. Also the strategies for verifying algorithms presented in Chapter 3 have only been roughly touched upon [KNR12]. Moreover, [KNR12] focuses on the small example of the Rotating Coordinator algorithm (see Section 5.1). In addition to the example of the Rotating Coordinator algorithm, this work presents details of all four case studies involved.

Earlier versions of the case studies have been subjects of student projects, a bachelor thesis, and a diploma thesis. All of these projects and work have been done under my supervision.

- Based on a model that has been worked out by myself, an earlier version of our Isabelle/HOL proof for the two safety properties of Distributed Consensus for the algorithm considered in Section 5.2 Validity and Agreement has been presented by Christina Rickmann in her Bachelor thesis submitted at TU-Berlin with the title ‘Formalisierung der Verifikation eines verteilten Konsens-Algorithmus mit Isabelle’ (‘Formal Verification of a Distributed Consensus Algorithm’).

- An earlier version of the Isabelle/HOL model and Isabelle/HOL proofs for the Paxos algorithm (see Section 5.3 have been developed in a student project at TU-Berlin. Models and proofs have been carried out by the students Christina Rickmann, Matthias Rost and Étienne Coulouma, and myself (Philipp Küfner).

- An earlier version of our model and proofs for the shared memory implementation for the Alpha-Framework (see Section 5.4) have been part of a student project at TU-Berlin. Models and proofs have been carried out by the students Benjamin Bisping, Liang Liang Ji, and myself (Philipp Küfner).

- My basic model and the proofs for the algorithm considered in Section 5.2 have been adapted and used for the verification of a broadcast-free variant of the mentioned algorithm (see Section 5.2.3). Parts of the proofs have been carried out in a diploma thesis by Qiang Jin submitted at TU-Berlin with the title ‘Formal Verification of a Broadcast-Free Consensus Algorithm in Isabelle’.

2. Modelling Distributed Algorithms

The term *model* is very abstract and of course can be used for many different applications. Therefore our first concern is to clarify our notion of a model. As explained in the introduction, our main goal is to develop new formal methods for proving the correctness of distributed algorithms. Generally, for showing that an algorithm actually does what it is supposed to do, we need a precise description of the algorithm and for doing formal proofs it is required that formal semantics are given for this description. Last but not least, it is necessary to have methods for reasoning over the given description. In the following, whenever we speak of a **formal model of an algorithm** we mean the formal unambiguous description of $\mathfrak{A}$. Moreover, if we speak of **our model** we mean the data structures and definitions introduced in this chapter (see Section 2.3) that enable us to define a formal model of an algorithm. Those data structures and definitions will make it possible to formally reason over the model. To apply a method for reasoning, the description of the algorithm must be formulated in terms of the method. Useful approaches for reasoning over our models are explained in Chapter 3.

For the next section it will be important to properly distinguish between the terms 'model', '(distributed) algorithm', 'distributed system' and 'specification'. In general by a **model** we mean an abstraction or a concept for something. As explained before, an abstraction for a certain algorithm becomes more concrete if we define it formally with the means introduced in this chapter but since it is no 'real world' implementation it still remains an abstraction (that is why we call it a model of the algorithm). A sequential **algorithm** is a description how a process will behave and analogously a distributed algorithm is a description how a set of processes behave within a **distributed system**. The distributed system comprises the set of processes and furthermore all parameters of the setting, e.g. whether processes may fail or how processes communicate. A **specification** can be either a specification of the algorithm or a specification of the requirements, i.e. of the properties the algorithm must exhibit. As explained, a specification of the algorithm can be given in terms of our model and will result in a formal model of the algorithm. The usual way to show correctness is to prove that the algorithm meets the specified requirements if the parameters of the system are as assumed. Therefore it must be possible not only to specify the algorithm in terms of our model but also the requirements and the parameters of the model to enable formal reasoning. Section 2.4 explains how our model supports the specification of requirements and how they are expressed as properties.

For our later formalisation, we integrated all parameters of the setting of an algorithm into the specification of the algorithm and therefore there is no formal difference between a distributed system and a distributed algorithm in our model. Hence, in our formal descriptions we only use the term *distributed algorithm*, which is then used synonymously to *distributed system*.

2.1. Level of Abstraction

Every model can only represent a subset of the real world. The level of abstraction describes how detailed this subset is. Most people expect from a *good* model that it exhibits the essential characteristics of the real world and hence captures the fundamental aspects that constitute the objective of the later analysis. Therefore the choice of the model depends on the objective which is to be analysed. In general one would like to avoid adding details to the model which do not concern this objective since this might hinder to find and examine the relevant parts of the model. Moreover, usually it is desired that the model is easy to understand. Hence, a higher level of abstraction that hides irrelevant details seems to be desirable. On the other hand, unforeseen side effects can be identified only, if the respective causes are modelled with enough details. Therefore the right level of abstraction is very fundamental for the usefulness of the model and this might be a trade-off between the level of detail and the 'easy-to-understand' performance of the model.

Verification means to prove that the (modelled) system meets the requirements. As already explained, the intention to do formal verification raises a further demand: the desired proof goals, i.e. the desired properties of the respective algorithm, must be formulated in terms of the model (otherwise it would be impossible to reason about the model in the theorem prover). Hence, the level of abstraction must be detailed enough to do formal proofs.

Most properties that matter in fault tolerant computing are of the kind 'for every execution of the algorithm it holds that ...'. Hence the language or logic used to express the properties as well as the model must comprehend the notion of an execution. Therefore it is necessary that the specification of the algorithm must define the single steps the algorithm performs. We use the term *Run* to capture one execution of the given algorithm and based on the terminology of Lamport (cf. [Lam02], [Lam94]) we use the term *Action* for the specification of an atomic transition a process can perform within a step (how actions can be defined will be explained in Section 2.3.3).

Of course in practice a deviation from the specification is possible in the case of failures. If failures are to be taken into account for a formal verification it is essential to specify the possible failures that may occur. Hence a failure model must be integrated into the model. Section 2.3.8 gives a summary of common failure models including the failure models that are used for the case studies in Chapter 5.

A further important part of a distributed system is the interprocess communication (ipc). For the correctness of an algorithm it might be very essential whether communication is synchronous or not.

In sum extending the different abstractions introduced in 2.1 the specification of a distributed system must include:

- the specification of the steps that are executed on the different processes
- the specification of the communication media
- the specification of the failure model
- (additional assumptions about the environment)

Commonly the level of abstraction varies for different components of a model. As an example consider the pseudo code, which is an often used means for illustrating how processes locally behave. Typically such a pseudo code will use directives like *SEND* or *RECEIVE* without a further description how these subroutines for the message infrastructure work (as an example see [CT96]).

Analogously to the four components enumerated above [GR06] names two basic abstractions that matter in a distributed system model: the process abstraction (describing the behaviour of the single local processes including failures etc.) and the communication abstraction (describing the way processes can communicate including the media that may allow message losses etc.). As an additional important abstraction [GR06] mentions the failure detector abstraction. Failure detectors can be regarded as local modules monitoring the crash status of all involved processes in the system.

Of course there are models and algorithms that do not make use of a failure detectors and there are models using different external components (e.g. a leader detection unit). Hence, we generalise this abstraction to an abstraction of the environment and so it can be seen as the context in which processes and communication take place.

Each of these abstractions must be chosen with respect to the needed level of detail and must be able to collaborate with each other to make the model consistent.

2.2. Related Approaches

The common way to present distributed algorithms is a pseudo-code-like representation based on a state-machine approach (cf. [Sch90], [Lyn96]).

This is explained in [GR06] by the lack of an appropriate distributed programming language. The argument in [GR06] is that notations of existing languages are to complicated and inventing a new one would require at least a book to explain it. But of course there are no formal semantics for pseudo-code.

Therefore the proofs given for the correctness of these algorithms often lack formal details. As a reason for this Charron-Boste and Merz mention the ‘scalability problem of current formal methods’ and propose to use the ‘HO-Model’ (introduced in [CBS07]). They claim this model will overcome the subtleness that commonly is inherent in the operations and assumptions that are used to describe distributed algorithms. [CBM09] shows that it is possible to adopt this model in a theorem proving environment. The HO-Model is designed for algorithms that proceed in rounds. Hence only round-based algorithms can be modelled within the framework of the HO-Model.

A more general approach for formal modelling is given by Leslie Lamport in [Lam94]. Lamport tries to overcome the lack of formal details by using his specification language Temporal Logic of Actions (TLA/TLA$^+$) for specifying distributed systems ([Lam02]). A formal proof based on a TLA$^+$ specification for the correctness of the distributed algorithm Disk Paxos can be found in [JM05]. This work adopts the ideas of modelling actions as predicates from [JM05].

2.3. Developing a Formal Model

Fundamentals for our approach are given in [FMN07] and [Fuz08]. They translate a given example of pseudo-code to formal transition rules. A main advantage of this approach is that it can be understood as a step of refinement: while the raw idea and intuition can be formulated via pseudo-code the transfer to transition rules forces the designer to define formal semantics for every operation. The approach in [FMN07] is primarily engineered and customised for the requirements given by the inspected algorithm (namely a Consensus algorithm given in [CT96]) but it shows many characteristics of common formal specification languages:

- It is based on predicate logic.
- Processes' local steps and all further possible steps are modelled by an interleaving semantics, i.e. a local step is mapped to a transition rule with well-defined pre- and post conditions for the global view.
- The global view on the system is given by sequences of global states called 'runs'.
- There is a designated set of initial states where every runs first state originates from.
- For every transition in a run from one state to its successor the transition rules are obeyed.

Notions of these characteristics are essential for formalism as abstract state machines (introduced in [Gur95]) and TLA$^+$ ([Lam02]). Moreover the approaches are similar enough to transfer models from [Fuz08] to TLA$^+$. Hence if needed a specification can be transferred from one model to the other very quickly.

The model in [Fuz08] is chosen as a basis for the formalisation in this work because it offers the following features:

- The fundamentals of the model (predicate logic) fit the input language of the theorem prover (First Order Logic and Higher Order Logic)
- It is focused on the problem, i.e. all details that are needed for the proofs are modelled and nothing more.
- A key contribution is that it can be used without formalising all details of a specification language as TLA$^+$ but is close enough to known formalism to be understood in the respective communities.
- It is an asynchronous model.

In an asynchronous system we have no upper bounds on message delay and process speed. There are different opinions about how realistic an asynchronous model is, because for the most practical applications programmers have to assume upper bounds and a system where some process is 10000 times slower than another does not seem realistic. Nevertheless, we develop a model for the asynchronous case because from a theoretical point of view problems in an asynchronous environment are more interesting. Since there are no assumptions on delay of processes and messages this can be seen as the most general case and hence all synchronous cases can be interpreted as subsets of the cases of the asynchronous case. In other words every algorithm that has been verified in an asynchronous environment will also work in a synchronous setting. Furthermore for practical application an extension of the model for synchrony can be done by adding respective assumptions for the bounds on message delay and process speed.

After establishing the model, we used the proofs given in [Fuz08] as a guideline for our proofs within the theorem prover. Particular attention in this work is paid to the differences between the paper proofs and the mechanical proofs within the theorem prover.

2.3.1. Runs and Temporal Logic

As explained before, verifying a system usually means to show that the system behaves as expected in every run. A run depicts the behaviour of a process in time. Hence, to prove that a system meets the specified requirements, it is necessary that the underlying model supports a notion of time to model the behaviour of the system along a time line. For our model we assume a discrete time line.

The formalisation in this work uses the abstractions introduced in [Fuz08] which are build on a set of inference rules that are used to describe the algorithm that is executed by a finite set of processes denoted by $\mathbb{P}$.

This can be interpreted as a *trace semantics* which is a simple kind of *observation semantics* and defines an observation as a visible sequence of states and actions [Gä02]. A trace is then commonly written as (cf. [Gä02]):

$$S_0 \overset{\mathcal{A}_0}{\rightarrow} S_1 \overset{\mathcal{A}_1}{\rightarrow} S_2 \overset{\mathcal{A}_2}{\rightarrow} \dots$$

which denotes that starting from an initial state S_0 the system transits to state S_1 by executing action $\mathcal{A}_0$, then executes $\mathcal{A}_1$ and reaches S_2 etc. [Gä02] mentions that trace semantics can also be used for inspecting behaviours of concurrent systems by extending states to *distributed states* or *global states* which are vectors of local states. Note that in this extension the state of a process (a *process state*) is a local state, which is only a part of the global state.

Hence for two processes p, p' working in parallel (without interaction) with traces $S_0 \overset{\mathcal{A}_0}{\rightarrow} S_1 \overset{\mathcal{A}_1}{\rightarrow} \dots$ and $S'_0 \overset{\mathcal{A}'_0}{\rightarrow} S'_1 \overset{\mathcal{A}'_1}{\rightarrow} \dots$ [Gä02] defines that

$$(S_0, S'_0) \overset{\mathcal{A}_0}{\rightarrow} (S_1, S'_0) \overset{\mathcal{A}'_0}{\rightarrow} (S_1, S'_1) \overset{\mathcal{A}_1}{\rightarrow} (S_2, S'_1) \overset{\mathcal{A}'_1}{\rightarrow} \dots$$

is a trace of the concurrent system where p and p' take turns. The depicted trace describes the special case of *fair interleaving* where p and p' alternately execute their actions. Of course different interleaving schedules are applicable. In [Fuz08] the already mentioned inference rules describe when and how transitions from one state to another may be executed. Hence the execution of an action corresponds to the application of some inference rule in [Fuz08] and the application of some inference rule means moving forward to the next point in time. Since the inference rules describe the algorithm this can be seen as the execution of one atomic step of the algorithm. Thus, starting from a dedicated set of initial system states, all possible evolutions of the system can be generated by applying a (possibly infinite) sequence of transition rules.

Note that having atomic steps between points in time implies a discrete time line. We observe again that the sequence of transition rules induce a sequence of global states that are called 'configurations' in [Fuz08]. Literature (e.g. [Lam02]) often uses the term states but since our work is based on the work of Fuzzati et al., we maintain their notion of configurations. A configuration can be seen as a snapshot of the system between two atomic steps. Hence, a configuration comprises all data that is relevant for computing a system step. [Fuz08] defines a configuration as the combination of all the modules (e.g. communication modules or failure detectors) that are present in the system and are used by the algorithm in addition to the state of the involved processes. This is obviously an extension of the vectors of local states [Gä02] uses, since our configurations contains an array of local states $\mathbf{S}$ but can additionally use modules for communication and other demands. Putting all together, a computation step is now a transition from one configuration to the next that respects the rules of the game, i.e. corresponds to one of the given inference rules. Hence we write a configuration as a vector containing the states of the processes and the states of the used modules. As Fuzzati [Fuz08] we allow to switch between horizontal and vertical notation for vectors to increase the readability.

Figure 2.1 depicts a general pattern for a transition: the states of modules $\mathbf{m}_1, \ldots, \mathbf{m}_\Gamma$ are represented by the values $\mathbf{M}1, \mathbf{M}2, \ldots, \mathbf{M}\Gamma$ and the state of the involved processes $\mathbf{s}$ is represented by $\mathbf{S}$ in the preceding configuration. Then the system transits to a configuration where the state of the modules has changed to $\mathbf{M}1', \mathbf{M}2', \ldots, \mathbf{M}\Gamma'$ and the state of the involved processes is $\mathbf{S}'$. So we assume that modules can be represented by some value $\mathbf{M}x$ and the state $\mathbf{S}$ of N involved processes is an array of the size N where each entry $\mathbf{S}(i)$ represents the state of one process. Note that $\mathbf{M}x$ and $\mathbf{S}(i)$ of course might be tuples or more complex data structures.

For a run of the algorithm the transition in Figure 2.1 must correspond to a transition rule as depicted in Figure 2.2. We adopt the ideas from [Fuz08] and define a configuration as follows:

Definition 2.3.1.1 (Configurations and Process States)
Let $\boldsymbol{M}1, \ldots, \boldsymbol{M}\Gamma$ be the modules relevant for the modelled distributed system and $\boldsymbol{S}$ be the state of all involved processes. Then a configuration $\mathfrak{C}$ is an assignment $\{\boldsymbol{S} \mapsto \boldsymbol{S}_{\mathfrak{C}}, \boldsymbol{M}1 \mapsto \boldsymbol{M}1_{\mathfrak{C}}, \boldsymbol{M}2 \mapsto \boldsymbol{M}2_{\mathfrak{C}}, \ldots, \boldsymbol{M}\Gamma \mapsto \boldsymbol{M}\Gamma_{\mathfrak{C}}\}$ of values $\boldsymbol{M}1_{\mathfrak{C}}, \ldots, \boldsymbol{M}\Gamma_{\mathfrak{C}}$ and $\boldsymbol{S}_{\mathfrak{C}}$ to $\boldsymbol{M}1, \ldots, \boldsymbol{M}\Gamma$ and $\boldsymbol{S}$.
By $\boldsymbol{S}_{\mathfrak{C}}$ we denote the array of process states in configuration $\mathfrak{C}$ and $\boldsymbol{S}_{\mathfrak{C}}(p_i)$ denotes the state of process $p_i \in \mathbb{P}$ in configuration $\mathfrak{C}$. The set of all process states is denoted by $\mathbb{S}$. Hence $\boldsymbol{S}_{\mathfrak{C}}$ can be interpreted as a mapping $\boldsymbol{S}_{\mathfrak{C}} : \mathbb{P} \to \mathbb{S}$

To project comfortably on different entries (state of modules) of a configuration and on entries of process states and states of modules, we introduce the following notations, which generalise the introduced notation for states:

Notation 2.3.1.2 (Accessing Tuples as Records)
Let $\mathfrak{C}$ be a configuration and $\boldsymbol{M}$ be a module, such that the state of $\boldsymbol{M}$ is represented in $\mathfrak{C}$. Moreover let the state of module $\boldsymbol{M}$ consist of the constituting parts $x_1, \ldots, x_k$. Then

- *$\boldsymbol{M}_{\mathfrak{C}}$ denotes the state of module $\boldsymbol{M}$ in $\mathfrak{C}$.*
- *$x_i \triangleright \boldsymbol{M}_{\mathfrak{C}}$ denotes the state of part x_i of module M in configuration $\mathfrak{C}$.*

For more complex data structures where a constituting part x of a module consists in turn of parts $y_1, \ldots, y_l$, we allow the extensive application of '$\triangleright$', i.e. we allow to write $y_r \triangleright x \triangleright \mathbf{M}_{\mathfrak{C}}$, to denote the state of y_r (which is part of the state of x, which is part of the state of module $\mathbf{M}$) in configuration $\mathfrak{C}$.

$$\begin{pmatrix} \mathbf{S} \\ \mathbf{M}1 \\ \mathbf{M}2 \\ \vdots \\ \mathbf{M}\Gamma \end{pmatrix} \rightarrow \begin{pmatrix} \mathbf{S'} \\ \mathbf{M}1' \\ \mathbf{M}2' \\ \vdots \\ \mathbf{M}\Gamma' \end{pmatrix}$$

Figure 2.1.: Transition in [Fuz08]

$$(\text{Rule})\ \frac{\text{Conditions on some of the configuration entries}}{\begin{pmatrix} \mathbf{S} \\ \mathbf{M}1 \\ \mathbf{M}2 \\ \vdots \\ \mathbf{M}\Gamma \end{pmatrix} \rightarrow \begin{pmatrix} \mathbf{S'} \\ \mathbf{M}1' \\ \mathbf{M}2' \\ \vdots \\ \mathbf{M}\Gamma' \end{pmatrix}}$$

Figure 2.2.: Transition rule ([Fuz08])

As already discussed a computation step is a transition from a configuration $\mathfrak{C}_i$ to a configuration $\mathfrak{C}_{i+1}$. How to get from one configuration to the next depends on the next step taken by the algorithm. Therefore the algorithm defines a step relation, i.e. let $\mathbb{C}$ denote the set of all configurations then the step relation $\rightarrow$ must be defined as a subset of $\mathbb{C} \times \mathbb{C}$. In [Fuz08] the step relation is given by the introduced inference rules. In this work we differentiate between steps taken by a process (*process-steps*) and steps where no dedicated actor is responsible for the step (*event-steps*). The cause for a process-step is always that a process executes some *process-action* while an event-step is caused by some *event* which happens unmotivated or at least without a dedicated process that triggered the event (e.g. the loss of a message while the message is transmitted). Formally a process-action is a predicate $PA : \mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \mathbf{bool}$ while an event is a predicate $EV : \mathbb{C} \times \mathbb{C} \rightarrow \mathbf{bool}$. A *Safe Algorithm* in our model is fully determined by a set of dedicated initial configurations, a set of process-actions and a set of events. We use the term *Safe Algorithm* because for a given *Safe Algorithm* $\mathfrak{A}$ we can prove that $\mathfrak{A}$ behaves *safe*, i.e., it does never violate given constrains (later we will formally introduce safety and liveness properties, see Section 2.4).

Definition 2.3.1.3 (Safe Algorithm)
A Safe Algorithm $\mathfrak{A}$ is a tuple $(\ \text{Init},\ \Phi,\ \Psi\ )$ *where* Init *is a (nonempty) set of (initial) configurations, Φ is a (finite) set of process-actions and Ψ is a (finite) set of events.*

Process-actions and events define the step relation of the algorithm.

Definition 2.3.1.4 (Steps)
Let $\mathfrak{A}$ be a (Safe) Algorithm with a set of process-actions Φ and Ψ set of events Ψ.

1. *A pair $(\ \mathfrak{C}_i,\ \mathfrak{C}_j\ )$ of configurations is called a* ***process-step*** *in $\mathfrak{A}$ if and only if $\exists \mathcal{A} \in \Phi\ .\ \exists p_k \in \mathbb{P}\ .\ \mathcal{A}(\mathfrak{C}_i, \mathfrak{C}_j, p_k)$.*
   We write $\mathfrak{C}_i \rightarrow_{\mathfrak{A},p_k:\mathcal{A}} \mathfrak{C}_j$ to denote a process-step in $\mathfrak{A}$ of process p_k from $\mathfrak{C}_i$ to $\mathfrak{C}_j$ with action $\mathcal{A}$.

2. *A pair $(\ \mathfrak{C}_i,\ \mathfrak{C}_j\ )$ of configurations is called an* ***event-step*** *in $\mathfrak{A}$ if and only if $\exists \mathcal{A} \in \Psi\ .\ \mathcal{A}(\mathfrak{C}_i, \mathfrak{C}_j)$*
   We write $\mathfrak{C}_i \rightarrow_{\mathfrak{A},\text{ev}:\mathcal{A}} \mathfrak{C}_j$ to denote an event-step in $\mathfrak{A}$ from $\mathfrak{C}_i$ to $\mathfrak{C}_j$ with action $\mathcal{A}$.

3. *A pair $(\ \mathfrak{C}_i,\ \mathfrak{C}_j\ )$ of configurations is called a* ***step*** *in $\mathfrak{A}$ if and only if $(\ \mathfrak{C}_i,\ \mathfrak{C}_j\ )$ is an event-step or a process-step.*
   We write $\mathfrak{C}_i \rightarrow_{\mathfrak{A}} \mathfrak{C}_j$ to denote a step in $\mathfrak{A}$ from $\mathfrak{C}_i$ to $\mathfrak{C}_j$.

In the following we omit $\mathfrak{A}$ and write only $\mathfrak{C}_i \rightarrow_{p_k:\mathcal{A}} \mathfrak{C}_j$, $\mathfrak{C}_i \rightarrow_{\text{ev}:\mathcal{A}} \mathfrak{C}_j$, and $\mathfrak{C}_i \rightarrow \mathfrak{C}_j$ if there is only one algorithm in the given context. The symbol $\rightarrow^*$ denotes the reflexive-transitive closure of $\rightarrow$ and the symbol $\rightarrow^\omega$ denotes an infinite sequence of $\rightarrow$.
We write $\mathfrak{C} \not\rightarrow$ if and only if from $\mathfrak{C}$ no further step is possible ($\nexists \mathfrak{C}' \in \mathbb{C}\ .\ \mathfrak{C} \rightarrow \mathfrak{C}'$). Then $\mathfrak{C}$ is a ***deadlock*** *in $\mathfrak{A}$.*

A run can now be defined as a (potentially infinite) sequence of configurations where each configuration and its successor are in the step relation. Formally we define a run as a mapping $R : \mathbb{T} \rightarrow \mathbb{C}$ where $\mathbb{T}$ denotes a countably infinite set of points in time. For convenience, in this work, we assume $\mathbb{T}$ to be the set of natural numbers ($\mathbb{T} = \mathbb{N}$) so that we can move one step forward in time by adding $+1$, backwards by adding -1 and 0 is the initial point in our time line. Speaking more general, it would suffice that our time domain is a total ordered set with a lower bound to apply our techniques. By Σ^ω we denote the set of all infinite sequences of configurations

$$\Sigma^\omega \triangleq \{\ S : \mathbb{T} \rightarrow \mathbb{C}\ \}\,.$$

Definition 2.3.1.5 (Infinite Runs)
Let $\mathfrak{A}$ be a (Safe) Algorithm with $\mathfrak{A} = (\ \text{Init},\ \Phi,\ \Psi\ )$, let $\rightarrow$ be the corresponding step relation. $R \in \Sigma^\omega$ is an infinite run of $\mathfrak{A}$ if and only if

1. $R(0) \in \text{Init}$

2. $\forall t \in \mathbb{T}\ .\ R(t) \rightarrow R(t+1)$

Therefore, $R(0)$ is the initial configuration of R, $R(1)$ the next configuration etc. Figure 2.3 shows a run R where in the initial configuration $R(0)$ action $\mathcal{A}_0$ is executed. By the execution of $\mathcal{A}_0$ the system transits to configuration $R(1)$ until $\mathcal{A}_1$ is executed and the system transits to configuration $R(2)$. Then consecutively actions $\mathcal{A}_1, \ldots, \mathcal{A}_{13}$ are executed and each $\mathcal{A}_i$ leads to configuration $R(i+1)$. Note that runs are sequences of configurations. Hence, for a given run it might be possible to deduce which action has been executed from configuration $R(i)$ to $R(i+1)$, but this information is not explicitly given by a run. This means that our model is strictly state-based.

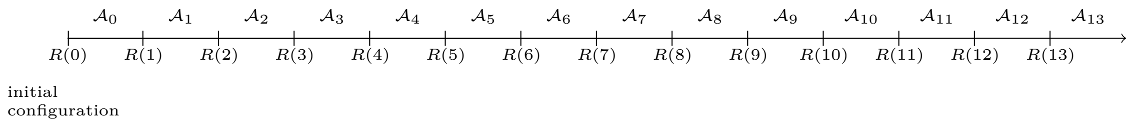

Figure 2.3.: Execution of actions $\mathcal{A}_0, \ldots, \mathcal{A}_{13}$ in Run R

Since the domain of a run R is the set of natural numbers, by definition a run is always infinite. To take finite runs into consideration we allow *final-stuttering* for our general definition of runs. A run R with final-stuttering reaches a deadlocked configuration in time t and for all successive configurations $t' \geq t$ we repeat the configuration $R(t)$ (hence we have $R(t') = R(t)$). Formally we define a predicate FinSt : $\mathbb{C} \times \mathbb{C} \to \mathbf{bool}$ for a given step relation $\to$:

$$\text{FinSt}(\mathfrak{C}_i, \mathfrak{C}_j) \triangleq (\mathfrak{C}_i = \mathfrak{C}_j) \wedge \mathfrak{C}_i \not\to$$

Now we are able to give a more general definition of runs:

Definition 2.3.1.6 (Runs)
Let $\mathfrak{A}$ be a (Safe) Algorithm with $\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi\ )$, let $\to$ be the corresponding step relation. $R \in \Sigma^\omega$ is a run of $\mathfrak{A}$ if and only if

1. $R(0) \in \text{Init}$
2. $\forall t \in \mathbb{T}\ .\ (R(t) \to R(t+1) \vee \text{FinSt}(R(t), R(t+1)))$

Runs($\mathfrak{A}$) *denotes the set of all runs of an algorithm $\mathfrak{A}$.*

Our definition of runs implies that a deadlocked configuration is repeated forever in a run:

Proposition 2.3.1.7 (Deadlocks last forever)
Let $R \in$ Runs($\mathfrak{A}$) be a run of a Safe Algorithm $\mathfrak{A}$ with a corresponding step relation $\to$ and $t \in \mathbb{T}$.
If R is deadlocked at time t ($R(t) \not\to$) then for all $t' \geq t$ we have $R(t') \not\to$ and $R(t') = R(t)$.

Proof.[1] $\hookrightarrow$ Isabelle[2]

[1] As explained in the introduction, full formal proofs of our propositions, lemmas, theorems, etc. are done in Isabelle/HOL. Therefore we will only give the reference to the Isabelle theory and theorem. Sometimes we will mention the main arguments for the proof.

[2] Isabelle/HOL theory: `DistributedAlgorithm.thy`, Lemma(s): `deadlockLastsForever`

From the definition of a deadlock and the definition of runs we obtain that If R is deadlocked at time t then we have

$$R(t+1) \not\to \text{ and } R(t+1) = R(t). \qquad \text{(DEADCONF)}$$

This is used in the inductional proof of Proposition 2.3.1.7. Note that it is often easier to use DEADCONF than the stronger version in Proposition 2.3.1.7 in similar inductional proofs. Therefore in our formal proof we provide DEADCONF as an additional lemma.

It is necessary to make further assumptions to prove certain properties. This means to reduce the set of considered executions to a subset that fulfills some defined constraints, e.g. that only a limited number of processes is allowed to crash. Therefore we extend the definition of a Safe Algorithm to *Algorithms* by adding a set of predicates $\mathcal{F}_{\text{ad}}$:

Definition 2.3.1.8 (Algorithm)
An Algorithm $\mathfrak{A}$ *is a tuple* (Init, Φ, Ψ, $\mathcal{F}_{\text{ad}}$) *where* (Init, Φ, Ψ) *is a Safe Algorithm and* $\mathcal{F}_{\text{ad}} \in \mathcal{P}\,(\Sigma^\omega \to \textbf{bool})$ *is a set of predicates on sequences of configurations.*

For Algorithms we introduce the notion of *Admissible Runs* which are the subset of runs where the predicates in $\mathcal{F}_{\text{ad}}$ hold:

Definition 2.3.1.9 (Admissible Runs)
An Admissible Run of an Algorithm $\mathfrak{A} =$ (Init, Φ, Ψ, $\mathcal{F}_{\text{ad}}$) *is a run* $R \in \text{Runs}(\mathfrak{A})$ *where* $P(R)$ *holds for all* $P \in \mathcal{F}_{\text{ad}}$. *Hence the set* $\text{Runs}_{\text{ad}}(\mathfrak{A})$ *is defined formally:*

$$\text{Runs}_{\text{ad}}(\mathfrak{A}) \triangleq \{\ R \in \text{Runs}(\mathfrak{A}) \ \mid\ \forall P \in \mathcal{F}_{\text{ad}}\ .\ P(R)\ \}$$

The different notions of Runs and Admissible Runs enable us to distinguish properties that hold for every run from properties that hold only under the assumptions in $\mathcal{F}_{\text{ad}}$. This will further be explained in Section 2.4.

Subsection 'Finite and Infinite Runs' will present alternative Modelling approaches for runs with special focus on the combination of finite and infinite runs.

Related Approaches Using Temporal Logic

We adopt the term 'action' from Leslie Lamport's Temporal Logic of Actions (TLA). Overall, our definitions for runs and steps have an obvious relation to temporal logic, e.g. TLA. In his Temporal Logic of Actions, Lamport refers to the assignments (configurations) as states and to the respective sequence of assignments (runs) as behaviours [Lam02]. Also our term 'step' has similar semantics in TLA.

We focus on the view of exactly one distributed system and are interested in properties that hold for every execution of this system. Recall that the term 'configuration' is used for the global state of the system, i.e. the current assignment of values to the relevant variables used by the system. On the contrary, Lamport's states refer to all variables, even to those, which are not used in the inspected system. Lamport ([Lam02]) proposes to use a relation `Next` which is typically a disjunction of actions that describe possible steps of the algorithm. Our work adopts the idea of actions in the way Jaskelioff and Merz ([JM05]) use actions in their formalisation of the

Disk Paxos [GL00] algorithm. This means that actions are predicates taking two configurations as parameters. An action $\mathcal{A}$ is true for two configurations c, c' (the value of $(\mathcal{A}\ c\ c')$ is true) if and only if a step from c to c' can be taken by executing action $\mathcal{A}$. Of course having two (finite) sets of actions (process-actions and events) is a special case of a disjunction of actions: Let $\mathfrak{A} = (\text{Init}, \{\ \mathcal{A}_1,\ \ldots,\ \mathcal{A}_k\ \}, \{\ \mathcal{A}_l,\ \ldots,\ \mathcal{A}_m\ \}, \mathcal{F}_{\text{ad}})$ be an algorithm. By definition of our step relation for two configurations $\mathfrak{C}_i, \mathfrak{C}_j \in \mathbb{C}$ the following holds:

$$\mathfrak{C}_i \to \mathfrak{C}_j \equiv (\exists p \in \mathbb{P}\ .\ \mathcal{A}_1(\mathfrak{C}_i, \mathfrak{C}_j, p) \vee \ldots \vee \mathcal{A}_k(\mathfrak{C}_i, \mathfrak{C}_j, p)) \vee \mathcal{A}_l(\mathfrak{C}_i, \mathfrak{C}_j) \vee \ldots \vee \mathcal{A}_m(\mathfrak{C}_i, \mathfrak{C}_j).$$

Hence our model implements the idea of Lamport's `Next` relation by using two sets of actions.

Enabledness

In TLA$^+$ there exists a predicate ENABLED, which is true for an action if and only if the action can be executed. Similar to TLA$^+$ we define predicates $\textbf{Enabled}_A$ and $\textbf{Enabled}_E$ to determine whether a process-action respectively an event can be executed:

Definition 2.3.1.10
Let $\mathfrak{A} = (\ \text{Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}}\ )$ *be an algorithm with* $\mathcal{A} \in \Phi$*,* $\hat{\mathcal{A}} \in \Psi$ *and* $\mathfrak{C}, \mathfrak{C}' \in \text{Runs}(\mathfrak{A})$*.*

$\textbf{\textit{Enabled}}_A : (\mathbb{C} \times \mathbb{C} \times \mathbb{P} \to \textbf{\textit{bool}}) \times \mathbb{C} \times \mathbb{P} \to \textbf{\textit{bool}}$ *is defined as follows:*

$$\textbf{\textit{Enabled}}_A(\mathcal{A}, \mathfrak{C}, p) \triangleq \exists \mathfrak{C}' \in \mathbb{C}\ .\ \mathcal{A}(\mathfrak{C}, \mathfrak{C}', p)$$

$\textbf{\textit{Enabled}}_E : (\mathbb{C} \times \mathbb{C} \to \textbf{\textit{bool}}) \times \mathbb{C} \to \textbf{\textit{bool}}$ *is defined as follows:*

$$\textbf{\textit{Enabled}}_E(\hat{\mathcal{A}}, \mathfrak{C}) \triangleq \exists \mathfrak{C}' \in \mathbb{C}\ .\ \hat{\mathcal{A}}(\mathfrak{C}, \mathfrak{C}')$$

To exclude runs in which some actions are always enabled, but never executed, the set of runs may then be restricted by fairness assumptions in $\mathcal{F}_{\text{ad}}$. Techniques to define fairness assumptions in terms of *Weak Fairness* and *Strong Fairness* are explained in [Lam02]. Examples for such fairness assumptions are given in our case studies in Sections 5.2 and 5.4.

Finite and Infinite Runs

As explained before, infinite sequences of configurations can be modelled as a mapping

$$R : \mathbb{N} \to \mathbb{C}$$

where $\mathbb{C}$ denotes the type of the configuration. We have also seen that problems occur if these sequences are only possibly but not necessarily infinite (e.g. as explained before if the system runs into deadlock and no successor configuration is reachable) and that we can solve these problems by introducing final stuttering. The work of Devillers, Griffioen and Müller [DGM97] analyses the advantages of alternative Modelling approaches for sequences of system states. Since the approach they call HOL-Fun is very close to our approach and the dual counterpart to this approach is (in some sense) the HOL-Sum, we give a short summary of the results they mention for their analyses.

An important issue is the ability to model infinite as well as finite sequences using the different approaches.

In the HOL-Fun approach sequences are modelled using partial functions of type $\mathbb{N} \rightharpoonup \mathbb{C}$. Functions are defined partially to allow finite sequences as follows: If the sequence is finite there is a number n where for all $m > n$ the function is not defined. Hence valid sequences must fulfil the following predicate ([DGM97]):

$$is\_sequence(s) = (\forall i.s(i) = \text{None} \Rightarrow s(i+1) = \text{None}) \tag{2.1}$$

(where $s(i) = \text{None}$ denotes that s is undefined for the argument i). Every operation on sequences must obey to return only sequences fulfilling this predicate. Hence some kind of normalisation is needed for every operation if it does not preserve the predicate *is_sequence*. Devillers et al. regard this as the main disadvantage of the HOL-Fun approach ([DGM97]).

The HOL-Sum approach is proposed in [PC96]. The definition in this approach is split in two sub definitions, one for the finite case and one for the infinite. The benefit of this alternative is that no additional normalisation procedure as for the HOL-Fun is necessary ([DGM97]). But due to the two sub definitions every operation must be implemented for both cases and hence the effort is doubled up for every implementation of a new operation on the sequences (([DGM97]). Therefore in our work we use a modification of the HOL-Fun approach: As explained before we allow 'stuttering steps' (analogously to [AS86] and [Cho94]) from configuration s to s' (steps where all variables of the system do not change and therefore $s = s'$ holds) if and only if the system is already terminated in state s. To understand why this is important for our model, assume an algorithm that is supposed to terminate in every run. If we omit the final stuttering option and allow only infinite runs this algorithm would render the set of (infinite) runs empty. As mentioned before, most of the considered properties are of the kind 'for every run R the following proposition holds ...'. Since our set of considered runs has become empty such a property is trivially true. Since often deadlocks are not intended this would be a fatal problem of the model, which is ruled out by the introduction of final stuttering. The following section analyses further but different problems that might occur when there are already inherent faults in the model and shows techniques how these problems and faults can be avoided.

2.3.2. Distribution and Consistency of our Model

In the field of verification the most fatal error that may occur is that the proof of correctness is error free but the model for the algorithm is wrong. The verification of the Paxos algorithm in [Fuz08] gives an example for this: Redoing the proofs of [Fuz08] we discovered that the defined inference rule for the delivery of a broadcast b was never enabled for any process but for the sender of b. Based upon this broadcast, processes were meant to decide a value v. The extensive proof of Agreement given by Fuzzati shows that if two processes decide values v_1, v_2 then $v_1 = v_2$ holds. As we see now, this property is actually trivial for the erroneous model since in every execution of a run only one process at all was able to decide a value.

Therefore using a theorem prover may guarantee faultless reasoning but every faultless reasoning is rendered useless if the underlying model of the algorithm does not reflect the intended implementation of the algorithm or does not respect the conditions of the assumed setting.

Of course it is not possible to rule out every error that might occur when the model for a concrete algorithm is developed. But in the following we introduce means to avoid unintended Modelling issues concerning the concept of distribution. Of course the concept of distribution is a crucial notion in the field of distributed algorithms and of course there are certain constraints, that must be respected by the algorithm. To rule out models which solve distributed problems but disregard the concept of distribution, this section defines a *Distributed Algorithm* as an algorithm that respects certain criteria which implement our notion of distribution.

Of course the idea of a distributed system is that processes work autonomously but can communicate by using defined communication mechanisms. We assume that when a process executes an action, it may read and manipulate its own state but not the state of other processes. Moreover, e.g. for communication, it might be necessary that a process accesses other modules of the configurations.

Let us consider how we can model distributed algorithms by the means introduced before. Figure 2.4 gives an example which will be used throughout this section to explain the introduced concepts. In this example every process p_i stores only one number $n \in \mathbb{N}$ in his process state. n represents the number of messages p_i has already processed. Hence, whenever p_i processes a message it has received, p_i increments n by one. If p_i has no message to process, p_i keeps flooding the network by sending its value n to all other processes.

Formally a message m is represented as a triple $(\ s,\ r,\ c\ )$ where $s, r \in \mathbb{P}$ are sender and receiver and $c \in \mathbb{N}$ is the content of the message. By $\mathcal{M}$ we denote the set of all messages $\mathcal{M} = \mathbb{P} \times \mathbb{P} \times \mathbb{N}$

We use a module $\mathbf{M} : \mathcal{M} \times \{\ \textbf{outgoing},\ \textbf{transit},\ \textbf{received}\ \} \rightarrow \mathbb{N}$ to keep track of all messages in our system. A message is **outgoing** after the sender of the message has put the message into its outbox. A message is in **transit** after it has left the outbox of the sender until it reached the inbox of the receiver and, of course, is **received** when it is in the inbox of the receiver. $\mathbf{M}(m, \textbf{outgoing})$ (respectively $\mathbf{M}(m, \textbf{transit})$, $\mathbf{M}(m, \textbf{received})$) returns how many copies of message m exist with status **outgoing** (respectively **transit**, **received**). The number of received messages $\mathbf{M}(m, \textbf{received})$ is decremented if the receiver of m has processed a message m. With $\mathcal{T}_{\text{ags}}$ we denote the set $\{\ \textbf{outgoing},\ \textbf{transit},\ \textbf{received}\ \}$.

Initially there are no messages in the system and hence for all $m \in \mathcal{M}$ and all $t \in \mathcal{T}_{\text{ags}}$ we have $\mathbf{M}(m, t) = 0$. Processes initially have processed no message and hence their initial process state is 0.

Figure 2.4 depicts the respective functions $\mathbf{S}_0 : \mathbb{P} \rightarrow \mathbb{N}$ and $\mathbf{M}_0 : \mathcal{M} \times \mathcal{T}_{\text{ags}} \rightarrow \mathbb{N}$ for the initial configuration and $\text{Init}_{\text{flood}}$ defines the respective set of initial configurations. Note that here we defined $\mathbf{S}_0$ and $\mathbf{M}_0$ explicitly. As usual (cf. Notation 2.3.1.2), for a configuration $\mathfrak{C} = \begin{pmatrix} \mathbf{S} \\ \mathbf{M} \end{pmatrix}$, we will later write $\mathbf{S}_{\mathfrak{C}}$ to denote the first entry $\mathbf{M}$ of $\mathfrak{C}$, i.e. the array of process states in $\mathfrak{C}$ and $\mathbf{M}_{\mathfrak{C}}$ to denote the second entry.

We introduce the concept of function updates for a more comfortable way to notate actions:

Definition 2.3.2.1 (Function update)
Let $f : \alpha \rightarrow \beta$ be a function and $a \in \alpha, b' \in \beta$.
Then $f[a := b'] = f'$ where $f' : \alpha \rightarrow \beta$ is defined as follows:

$$f'(x) = \begin{cases} b' & , \textit{if } x = a \\ f(x) & , \textit{else} \end{cases}$$

Note that we can also use nested function update: a function update $(f[a_1 := b'_1])[a_2 := b'_2]$ returns by definition a function f'' with

$$f''(x) = \begin{cases} b'_1 & , \text{if } x = a_1 \wedge a_1 \neq a_2 \\ b'_2 & , \text{if } x = a_2 \\ f(x) & , \text{else} \end{cases}$$

In our example the set of messages a process p_i sends is constructed by the function $\mathit{New}_{\text{msgs}} : \mathbb{P} \times \mathbb{N} \rightarrow \mathcal{P}(\mathcal{M})$. $\mathit{New}_{\text{msgs}}(p_i, n)$ returns a set of messages which contains a message m for each $p_j \in \mathbb{P}$ such that the sender of m is p_i and the content is n and the receiver of m is p_j. In addition we define a subaction $\mathit{Insert}_{\text{msgs}}$, i.e. a predicate $\mathit{Insert}_{\text{msgs}}$ that is used by some action to assert that messages are inserted from one configuration to the next. Hence, $\mathit{Insert}_{\text{msgs}}(\mathfrak{C}, \mathfrak{C}', \mathit{new})$ is true if and only if for all messages in the set *new* the value $\mathbf{M}_{\mathfrak{C}}(m, \textbf{outgoing})$ is incremented by one to $\mathfrak{C}'$ and all other values of $\mathbf{M}_{\mathfrak{C}'}(m', t)$ are equal to $\mathbf{M}_{\mathfrak{C}}(m', t)$.

We define two process-actions:

***ProcessMsg*:** If there is a message m_r for a process p_i which is **received** then p_i processes the message by incrementing its state by one and decrementing the received copies of m by one to indicate that m has been processed.

***SendToAll*:** If there is no message to process for $p_i \in \mathbb{P}$, p_i is able to send messages. As described, the sending of messages uses $\mathit{Insert}_{\text{msgs}}$ and the constructor $\mathit{New}_{\text{msgs}}$. All process states remain unchanged by this action.

Furthermore the communication infrastructure comprises two events:

***SendMsg*:** *SendMsg* puts a message m from the status **outgoing** to **transit**, i.e. decrements the number of copies of m that are **outgoing** by one and respectively increments the number of copies that have status **transit**.

***Deliver*:** This action works as *SendMsg* but moves a copy of m from **transit** to **received**.

Let us now inspect the details of the process-action *ProcessMsg*. As we would expect from a process-action this action works only on values the executing process p_i can read and write, i.e. p_i's own process state and its received messages and no other storages which p_i would not able to access. But which values p_i exactly is able to read and write is very implicit in the model. Moreover, only the name **outgoing** of the status of a message m tells us that p_i can not read

Initial function definitions:

$$\mathbf{S}_0 : \mathbb{P} \to \mathbb{N} \mid \mathbf{S}_0(p_i) \triangleq 0 \qquad\qquad \mathbf{M}_0 : \mathcal{M} \times \mathcal{T}_{\text{ags}} \to \mathbb{N} \mid \mathbf{M}_0(m,t) \triangleq 0$$

Model definitions:

$$\begin{aligned}
\text{Init}_{\text{flood}} &\triangleq \left\{ \begin{pmatrix} \mathbf{S}_0 \\ \mathbf{M}_0 \end{pmatrix} \right\} \\
\mathit{ProcessMsg}(\mathfrak{C}, \mathfrak{C}', p_i) &\triangleq \exists m_r \in \mathcal{M} \,.\, \exists p_s, n \,.\, m_r = (\ p_s,\ p_i,\ n\ ) \wedge \mathbf{M}_{\mathfrak{C}}(m_r, \textbf{received}) > 0 \\
&\quad \wedge \mathbf{M}_{\mathfrak{C}'} = \mathbf{M}_{\mathfrak{C}}[(m_r, \textbf{received}) := \mathbf{M}_{\mathfrak{C}}(m_r, \textbf{received}) - 1] \\
&\quad \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{\mathfrak{C}}(p_i) + 1] \\
\mathit{New}_{\text{msgs}}(p_s, n) &\triangleq \{\ m \in \mathcal{M} \ \mid\ \exists p_r \in \mathbb{P} \,.\, m = (p_s, p_r, n)\ \} \\
\mathit{Insert}_{\text{msgs}}(\mathfrak{C}, \mathfrak{C}', \mathit{new}) &\triangleq \left(\mathbf{M}_{\mathfrak{C}'}(m,t) = \begin{cases} \mathbf{M}_{\mathfrak{C}}(m,t) + 1 & \text{, if } (t = \textbf{outgoing} \wedge\ m \in \mathit{new}) \\ \mathbf{M}_{\mathfrak{C}}(m,t) & \text{, else} \end{cases} \right) \\
\mathit{SendToAll}(\mathfrak{C}, \mathfrak{C}', p_i) &\triangleq \{\ m \in \mathcal{M} \ \mid\ \exists p_s, n \,.\, m = (\ p_s,\ p_i,\ n\ ) \wedge \mathbf{M}_{\mathfrak{C}}(m, \textbf{received}) > 0\ \} = \emptyset \\
&\quad \wedge \mathit{Insert}_{\text{msgs}}(\mathfrak{C}, \mathfrak{C}', \mathit{New}_{\text{msgs}}(p_i, \mathbf{S}_{\mathfrak{C}}(p_i))) \\
&\quad \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}} \\
\mathit{SendMsg}(\mathfrak{C}, \mathfrak{C}') &\triangleq \exists m_o \,.\, \mathbf{M}(m_o, \textbf{outgoing}) > 0 \\
&\quad \wedge \mathbf{M}_{\mathfrak{C}'} = (\mathbf{M}_{\mathfrak{C}}[(m_o, \textbf{outgoing}) := \mathbf{M}_{\mathfrak{C}}(m_o, \textbf{outgoing}) - 1]) \\
&\qquad\qquad [(m_o, \textbf{transit}) := \mathbf{M}_{\mathfrak{C}}(m_o, \textbf{transit}) + 1] \\
\mathit{Deliver}(\mathfrak{C}, \mathfrak{C}') &\triangleq \exists m_t \,.\, \mathbf{M}(m_t, \textbf{transit}) > 0 \\
&\quad \wedge \mathbf{M}_{\mathfrak{C}'} = (\mathbf{M}_{\mathfrak{C}}[(m_t, \textbf{transit}) := \mathbf{M}_{\mathfrak{C}}(m_t, \textbf{transit}) - 1]) \\
&\qquad\qquad [(m_t, \textbf{received}) := \mathbf{M}_{\mathfrak{C}}(m_t, \textbf{received}) + 1] \\
\Phi_{\text{flood}} &\triangleq \{\ \mathit{SendToAll},\ \mathit{ProcessMsg}\ \} \\
\Psi_{\text{flood}} &\triangleq \{\ \mathit{SendMsg},\ \mathit{Deliver}\ \} \\
\mathcal{F}_{\text{flood}} &\triangleq \emptyset \\
\mathfrak{A}_{\text{flood}} &\triangleq (\ \text{Init}_{\text{flood}},\ \Phi_{\text{flood}},\ \Psi_{\text{flood}},\ \mathcal{F}_{\text{flood}}\ )
\end{aligned}$$

Figure 2.4.: Simple Algorithm $\mathfrak{A}_{\text{flood}}$

the contents of m as long as there is no **received** copy of m. Formally nothing prevents us up to now from defining a process-action $\mathit{ProcessMsg}^{\text{omnis}}$ where p_i already increments its process state for a message m that is still in the outgoing buffer of the sender:

$$\begin{aligned}
\mathit{ProcessMsg}^{\text{omnis}}(\mathfrak{C}, \mathfrak{C}', p_i) &\triangleq \exists m_r \in \mathcal{M} \,.\, \exists p_s, n \,.\, m_r = (\ p_s,\ p_i,\ n\ ) \wedge \mathbf{M}_{\mathfrak{C}}(m_r, \textbf{outgoing}) > 0 \\
&\quad \wedge \mathbf{M}_{\mathfrak{C}'} = \mathbf{M}_{\mathfrak{C}}[(m_r, \textbf{outgoing}) := \mathbf{M}_{\mathfrak{C}}(m_r, \textbf{outgoing}) - 1] \\
&\quad \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{\mathfrak{C}}(p_i) + 1]
\end{aligned}$$

Moreover, the 'omniscient' action $\mathit{ProcessMsg}^{\text{omnis}}$ changes the number of **outgoing** copies. Obviously, this is not intended for the given example and hence it is modelled as described by Figure 2.4. But the required global view from which a process-action is defined obviously can cause unintended modelling of access to parts of configurations the executing process is not able to read. This yields the need for means that enable us to define a more explicit separation of things a process is able to access and resources it can either only access indirectly or never.

To limit the scope a process is allowed to read and write, we define the notion of a *Localview* of a process. The Localview of a process can be interpreted as the variables and resources a process can access. Of course this Localview is part of the distributed system and therefore must be represented in a configuration. By $\mathbb{L}$ we denote the set of all Localviews and define a function

$$\text{C2LV} : \mathbb{C} \times \mathbb{P} \rightarrow \mathbb{L}$$

which returns for a configuration $\mathfrak{C}$ and process p_i the restricted view of p_i on $\mathfrak{C}$. This function varies for each algorithm and hence is a parameter for the definition of a distributed algorithm. In general, one essential part of a Localview of a process p_i is the (local) process state of p_i. Hence, we assume the Localview of a process p_i in a configuration $\mathfrak{C}$ to be a vector

$$(\mathbf{S}_{\mathfrak{C}}(p_i), \mathbf{x}_1(\mathfrak{C}), \mathbf{x}_2(\mathfrak{C}), \ldots, \mathbf{x}_\kappa(\mathfrak{C})) \tag{Localview}$$

where $\mathbf{S}_{\mathfrak{C}}(p_i)$ is the process state of p_i in $\mathfrak{C}$ and $x_1, \ldots, x_\kappa$ are functions over configurations. We overload the function $\mathbf{S}$ and denote by $\mathbf{S}_{lv}$ the process state of a Localview lv (which is the first entry of lv, cf. Localview above).

For consistency of the model, a process-action $\mathcal{A} \in \Phi$ executed by a process p_i may only read and affect parts of the configuration which p_i can access in its Localview. Hence, for a step $\mathfrak{C} \rightarrow_{p_i:\mathcal{A}} \mathfrak{C}'$ there must be $lv, lv' \in \mathbb{L}$ such that lv is the Localview of p_i in $\mathfrak{C}$, lv' is the Localview of p_i in $\mathfrak{C}'$ and nothing else has changed from $\mathfrak{C}$ to $\mathfrak{C}'$ but only the parts that p_i can access (this stipulation is formally defined later with equation LocalviewRespect).

Note that from a designer point of view it is much more convenient to define a process-action $\mathcal{A}$ by describing how $\mathcal{A}$ affects the Localview of a process (this will later be illustrated by examples and is further explained in Section 2.3.3). Hence, instead of defining a global action $\mathcal{A}$ which can be executed by a process p_i, we might want to definea predicate $a^{\downarrow}_{\mathcal{A}} : \mathbb{L} \times \mathbb{L} \rightarrow \mathbf{bool}$ that describes only the local behaviour of the action because a process action only performs local changes. Since $a^{\downarrow}_{\mathcal{A}}$ is supposed to represent of $\mathcal{A}$, $\mathfrak{C} \rightarrow_{p_i:\mathcal{A}} \mathfrak{C}'$ must imply $a^{\downarrow}_{\mathcal{A}}(lv, lv')$ with $lv = \text{C2LV}(\mathfrak{C}, p_i)$ and $lv' = \text{C2LV}(\mathfrak{C}', p_i)$ and vice versa. We call such a predicate $a^{\downarrow}_{\mathcal{A}}$ the *non-lifted* version of $\mathcal{A}$ and $\mathcal{A}$ the lifted version of $a^{\downarrow}_{\mathcal{A}}$. Note the difference between the types of $a^{\downarrow}_{\mathcal{A}}$ and $\mathcal{A}$: $a^{\downarrow}_{\mathcal{A}} : \mathbb{L} \times \mathbb{L} \rightarrow \mathbf{bool}$ is a predicate over Localviews, $\mathcal{A} : \mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \mathbf{bool}$ is a predicate that takes two configurations and a process.

To define the relation between $a^{\downarrow}_{\mathcal{A}}$ and $\mathcal{A}$ formally, we need a further function that embeds the Localview of a process back into a configuration. Note that the data of a Localview is only a subset of the information of the corresponding configuration. Hence, additional information must be provided for such an embedding. Let lv' be the Localview of process p_i. To extend lv' to a configuration we need the information that lv' is p_i's Localview and how the rest of the configuration should be arranged. During a process-step of p_i from a configuration $\mathfrak{C}$ to a configuration $\mathfrak{C}'$ only the data of p_i's Localview can be changed. Hence, it suffices to extract the rest of the configuration $\mathfrak{C}$ to compute the configuration $\mathfrak{C}'$. Therefore we define a function

$$\mathrm{LV2C} : \mathbb{L} \times \mathbb{P} \times \mathbb{C} \to \mathbb{C}$$

that embeds a Localview of a given process into a configuration and returns the resulting configuration. Note that obviously the domain of LV2C differs from the co-domain of C2LV and furthermore C2LV is not injective. Therefore LV2C is not the inverse function of C2LV. Hence, it is not possible to derive LV2C from C2LV directly. Thus, LV2C must be given as a further parameter in our definition of a Distributed Algorithm. For our example (Figure 2.4) the variables a process p_i can read, are the messages p_i has received (its *inbox*) and its program state. What p_i is able to do, is: put messages into its outbox, change its process state, and delete a processes message from its inbox. Let us first define a function $\mathrm{inbox}_{\mathfrak{C}}^{p_i}$ that extracts the inbox of p_i from a configuration $\mathfrak{C}$. We model the inbox analogously to the module $\mathbf{M}$, hence for a given message m the function $\mathrm{inbox}_{\mathfrak{C}}^{p_i} : \mathcal{M} \to \mathbb{N}$ returns the number of copies of m the process p_i got in his inbox. Therefore we can define $\mathrm{inbox}_{\mathfrak{C}}^{p_i}$ as follows:

$$\mathrm{inbox}_{\mathfrak{C}}^{p_i}(m) \triangleq \begin{cases} \mathbf{M}_{\mathfrak{C}}(m, \mathit{received}) & \text{, if } \exists p_s \,.\, \exists n \,.\, m = (p_s, p_i, n) \\ 0 & \text{, else} \end{cases}$$

Let us assume p_i is not able to see messages it has sent before. We can define $\mathrm{C2LV}_{\,\mathrm{flood}}$ as:

$$\mathrm{C2LV}_{\,\mathrm{flood}}(\mathfrak{C}, p_i) \triangleq (\mathbf{S}_{\mathfrak{C}}(p_i), \mathrm{inbox}_{\mathfrak{C}}^{p_i}, \emptyset, p_i) \qquad (\mathrm{C2LV}_{\,\mathrm{flood}})$$

Hence, for our example a Localview lv is a vector $(\mathbf{S}, \mathbf{In}, \mathbf{Out}, \mathbf{ID})$ where $\mathbf{S}$ is the process state of the Localview, $\mathbf{In}$ is the inbox (modelled as a function as described) and $\mathbf{Out}$ is the outbox, which is modelled as an (initially empty) set of messages. Note that the decision to use a set for the outbox implies a process is not able to send more than one copy of a message at once). Since process p_i has to set itself as the sender when generating messages, we use an extra field $\mathbf{ID}$ to store the identity of p_i (note that the executing process is not a parameter for the non-lifted versions of actions). Usually (if we do not consider a Byzantine process model) a process is not allowed to change its id.

As for configurations we denote the state entry of a Localview lv by $\mathbf{S}_{lv}$, the inbox entry by $\mathbf{In}_{lv}$, the outbox entry by $\mathbf{Out}_{lv}$ and the id entry by $\mathbf{ID}_{lv}$.

Now we can define non-lifted versions of *SendToAll* and *ProcessMsg*(cf. Figure 2.4):

$$\begin{aligned} \mathit{ProcessMsg}^{\downarrow}(lv, lv') &\triangleq \exists m_r \in \mathcal{M} \,.\, \mathbf{In}_{lv}(m_r) > 0 \\ &\quad \wedge lv' = (\mathbf{S}_{lv} + 1, \mathbf{In}_{lv}[m_r := \mathbf{In}_{lv}(m_r) - 1], \mathbf{Out}_{lv}, \mathbf{ID}_{lv}) \\ \mathit{SendToAll}^{\downarrow}(lv, lv') &\triangleq \forall m_r \in \mathcal{M} \,.\, \mathbf{In}_{lv}(m_r) = 0 \\ &\quad \wedge lv' = (\mathbf{S}_{lv}, \mathbf{In}_{lv}, \mathit{New}_{\mathrm{msgs}}(p_i, \mathbf{S}_{lv}), \mathbf{ID}_{lv}) \end{aligned}$$

Note how simple the definition of local transitions is compared to the long-winded lifted actions in Figure 2.4, which need the extra definition of $\mathit{Insert}_{\mathrm{msgs}}$.

To re-embed the Localview lv' of a process p_i in a configuration $\mathfrak{C}$ for $\mathfrak{A}_{\mathrm{flood}}$ we have to define the function LV2C$_{\text{flood}}$. First let us define how the message module $\mathbf{M}$ will change after a step from a Localview lv to lv'. On the one hand we have to increase the values of the messages p_i added to its outbox. To be sure that p_i does not sail under false colours, we increase the **outgoing** values only for messages were p_i is indeed the sender of the message. Furthermore, we have to delete the messages that p_i has already processed. To determine the right messages, we have to consider only **received** messages where p_i is the receiver of the message. For these messages we apply the values from $\mathbf{In}_{lv}$. Note that the first update concerns only **outgoing** messages while the second deals with **received** messages. Therefore there can be no interference between both updates. We define a function $\hat{\mathbf{M}}^{p_i}_{lv',\mathfrak{C}} : \mathcal{M} \times \mathcal{T}_{\text{ags}} \to \mathbb{N}$ that returns $\mathbf{M}_{\mathfrak{C}}$ updated by the information of a given Localview lv', which is assumed to be the (updated) Localview of process p_i after p_i executed an action in $\mathfrak{C}$:

$$\hat{\mathbf{M}}^{p_i}_{lv',\mathfrak{C}}(m,t) \triangleq \begin{cases} \min(\mathbf{In}_{lv'}(m), \mathbf{M}_{\mathfrak{C}}(m, \textbf{received})) & \text{, if } t = \textbf{received} \\ & \quad \wedge\ (\exists p_s \in \mathbb{P} \,.\, \exists n \,.\, m = (\ p_s,\ p_i,\ n\ )) \\ \mathbf{M}_{\mathfrak{C}}(m,t) + 1 & \text{, if } (t = \textbf{outgoing} \wedge m \in \mathbf{Out}_{lv'}) \\ & \quad \wedge\ (\exists p_r \in \mathbb{P} \,.\, \exists n \,.\, m = (\ p_i,\ p_r,\ n\ )) \\ \mathbf{M}_{\mathfrak{C}}(m,t) & \text{, else} \end{cases}$$

$$\text{LV2C}_{\text{flood}}(lv', p_i, \mathfrak{C}) \triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{lv'}] \\ \hat{\mathbf{M}}^{p_i}_{lv',\mathfrak{C}} \end{pmatrix} \tag{LV2C$_{\text{flood}}$}$$

Note that the use of min in $\hat{\mathbf{M}}^{p_i}_{lv',\mathfrak{C}}$ makes it impossible for p_i to add **received** messages on its own authority. p_i is only able to process (respectively delete) messages. Note further that by the definition of C2LV$_{\text{flood}}$ and LV2C$_{\text{flood}}$ a process p_i can see its identity but it is not able to change it, i.e. it can change it locally in his own scope and view for the execution of one action but this change will not be globally applied. Hence, the use of LV2C and C2LV enable us to model *read-only* values. Moreover, our notion of the outbox implements the concept of a *blind write*: although p_i is not able to see (and possibly) change its former outgoing messages, p_i can write and add new messages. We see that there are two main advantages that come with the use of non-lifted actions:

- Non-lifted actions describe local changes concisely and therefore are better readable.
- The use of C2LV and LV2C enable us to introduce some notion of read and write privileges for global variables, i.e. enable us to appoint which variables a process may only read, which variables it is also allowed to write and which variable it can also modify.

Of course, the embedding of a Localview of process p_i in a configuration $\mathfrak{C}$ back into $\mathfrak{C}$ for p_i must result in the configuration c and hence, we assume for C2LV and LV2C that for all $p_i \in \mathbb{P}$ and $\mathfrak{C} \in \mathbb{C}$:

$$\text{LV2C}(\text{C2LV}(\mathfrak{C}, p_i), p_i, \mathfrak{C}) = \mathfrak{C} \tag{locViewProp1}$$

Obviously this holds for our example C2LV$_{\text{flood}}$ and LV2C$_{\text{flood}}$. Note that we do not explicitly require C2LV(LV2C($\mathfrak{C}, p_i, lv$)$, p_i$) $= lv$, since for our model it is only relevant to get from a configuration to a Localview and back to a configuration (see Figure 2.5). Later, we will give examples where we actually have $\mathrm{C2LV}(\mathrm{LV2C}(\mathfrak{C}, p_i, lv), p_i) \neq lv$.

Having C2LV and LV2C we are now able to formally express what it means for a process-actionto respect the Localview of processes. We stipulate that for every process-action $\mathcal{A}$ there must be a non-lifted version $a^{\downarrow}_{\mathcal{A}}$ of $\mathcal{A}$ such that for all $\mathfrak{C}, \mathfrak{C}' \in \mathbb{C}$ and $p_i \in \mathbb{P}$ the following holds:

$$(\mathfrak{C} \rightarrow_{p_i:\mathcal{A}} \mathfrak{C}') \quad \leftrightarrow \quad \big(\exists lv' \,.\, a^{\downarrow}_{\mathcal{A}}(\mathrm{C2LV}(\mathfrak{C}, p_i), lv') \wedge \mathfrak{C}' = \mathrm{LV2C}(lv', p_i, \mathfrak{C})\big) \qquad \text{(LocalviewRespect)}$$

Intuitively this means: There is a step $\mathfrak{C} \rightarrow_{p_i:\mathcal{A}} \mathfrak{C}'$ if and only if we can find a Localview lv' such that $\mathfrak{C}'$ is the embedding of lv' for p_i in $\mathfrak{C}$ and a 'non-lifted step' with $a^{\downarrow}_{\mathcal{A}}$ is possible from $\mathrm{C2LV}(\mathfrak{C}, p_i)$ (which denotes the Localview of p_i in $\mathfrak{C}$) to lv'. Figure 2.5 illustrates the concept of a non-lifted action: instead of computing the step for an action $\mathcal{A}$ of a process p_i on the level of configurations we compute the Localview lv of p_i by using the function C2LV. Then we compute the local changes processed by executing the non-lifted action $a^{\downarrow}_{\mathcal{A}}$ resulting in lv'. Then we embed lv' back into $R(t)$ and the result we get is $R(t+1)$.

We define a function $\mathrm{Lift} : (\mathbb{L} \times \mathbb{L} \rightarrow \textbf{bool}) \rightarrow (\mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \textbf{bool})$ that returns the lifted version of a non-lifted action by giving the following abstraction:

Definition 2.3.2.2 (Lift)
For an arbitrary non-lifted action $a^{\downarrow}$ *let* $\mathrm{lf}_{a^{\downarrow}} : \mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \textbf{\textit{bool}}$ *denote the following function:*

$$\mathrm{lf}_{a^{\downarrow}}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq \big(\exists lv' \,.\, a^{\downarrow}(\mathrm{C2LV}(\mathfrak{C}, p_i), lv') \wedge \mathfrak{C}' = \mathrm{LV2C}(lv', p_i, \mathfrak{C})\big)$$

Then $\mathrm{Lift} : (\mathbb{L} \times \mathbb{L} \rightarrow \textbf{\textit{bool}}) \rightarrow (\mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \textbf{\textit{bool}})$ *is defined as:*

$$\mathrm{Lift}(a^{\downarrow}) \triangleq \mathrm{lf}_{a^{\downarrow}}$$

With this definition of Lift, equation LocalviewRespect can be rewritten as: for every process-action $\mathcal{A}$ there must be a non-lifted version $a^{\downarrow}_{\mathcal{A}}$ of $\mathcal{A}$ such that $\mathcal{A} = \mathrm{Lift}(a^{\downarrow}_{\mathcal{A}})$.

Note that for our example we already gave an equivalent non-lifted version for each $\mathcal{A} \in \Phi_{\text{flood}}$ (cf. $\mathit{ProcessMsg}^{\downarrow}, \mathit{SendToAll}^{\downarrow}$).

$$\begin{tikzcd} \mathrm{R(t)} \arrow[r, "\longrightarrow_{p_i:\mathcal{A}}", phantom] \arrow[d, "\mathrm{C2LV}(R(t),p_i)"'] & \mathrm{R(t+1)} \\ lv \arrow[r, "\rightarrow_{a^{\downarrow}_{\mathcal{A}}}", phantom] & lv' \arrow[u, "\mathrm{LV2C}(lv',p_i,R(t))"'] \end{tikzcd}$$

Figure 2.5.: Configurations, Localviews, C2LV and LV2C

To reflect processes' local state transitions and to circumvent process states of other processes to be manipulated, we stipulate that LV2C satisfies the following equation for all $lv \in \mathbb{L}, p_i, p_j \in$

$\mathbb{P}, \mathfrak{C} \in \mathbb{C}$:

$$\mathbf{S}_{\text{LV2C}(lv,p_i,\mathfrak{C})}(p_j) = \begin{cases} \mathbf{S}_{lv} & \text{, if } p_j = p_i \\ \mathbf{S}_{\mathfrak{C}}(p_j) & \text{, else} \end{cases} \tag{locViewProp2}$$

Since our example LV2C $_{\text{flood}}$ only uses function update at p_i, LV2C $_{\text{flood}}$ respects locViewProp2.

Regarding only process states, of course we want that the computation of a step by a process p_i may not depend on other processes' states but on the process state of p_i. Until now, it would still be possible to define C2LV $_{\text{flood}}$ as a function which maps states of all processes to the entries of the Localview and therefore make it possible for p_i to use information from other processes for its local computation.

To restrain such constructions by an assertion NoRemoteAccess, we define a predicate LocallyEqual : $\mathbb{C} \times \mathbb{C} \times \mathbb{P} \to \mathbf{bool}$. For two configurations $\mathfrak{C}, \mathfrak{C}'$ and a process $p_i \in \mathbb{P}$ we define:

$$\begin{aligned} \text{LocallyEqual}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq\ & \mathbf{S}_{\mathfrak{C}}(p_i) = \mathbf{S}_{\mathfrak{C}'}(p_i) \\ & \wedge (\forall n \in \{\ n \in \mathbb{N} \ \mid\ 1 \leq n \leq \Gamma\ \}\ .\ \mathbf{M}n_{\mathfrak{C}} = \mathbf{M}n_{\mathfrak{C}'}) \end{aligned}$$

where $\mathbf{M}n_{\mathfrak{C}_k}$ denotes the state of the n-th module of the system in configuration $\mathfrak{C}_k$ and Γ is the number of modules in the system. Hence two configurations $\mathfrak{C}, \mathfrak{C}' \in \mathbb{C}$ are LocallyEqual for a process $p_i \in \mathbb{P}$ as long as $\mathfrak{C}$ and $\mathfrak{C}'$ only differ in process states of $p_j \neq p_i$ (note that other module states must also be equal). Now we are able to formally define NoRemoteAccess. For all $p_i \in \mathbb{P}$, for all $\mathfrak{C}, \mathfrak{C}', \mathfrak{C}_1, \mathfrak{C}'_1 \in \mathbb{C}$ and for all $\mathcal{A} \in \Phi$ the following assertion must hold:

$$\mathcal{A}(\mathfrak{C}, \mathfrak{C}', p_i) \wedge \text{LocallyEqual}(\mathfrak{C}, \mathfrak{C}_1, p_i) \Rightarrow \mathcal{A}(\mathfrak{C}_1, \mathfrak{C}'_1, p_i) \tag{NoRemoteAccess}$$

where $\mathfrak{C}'_1 \triangleq \begin{pmatrix} \mathbf{S}_2 \\ \mathbf{M}1_{\mathfrak{C}'} \\ \mathbf{M}2_{\mathfrak{C}'} \\ \vdots \\ \mathbf{M}\Gamma_{\mathfrak{C}'} \end{pmatrix}$ and $\mathbf{S}_2(p_j) = \begin{cases} \mathbf{S}_{\mathfrak{C}'}(p_j) & \text{, if } p_j = p_i \\ \mathbf{S}_{\mathfrak{C}_1}(p_j) & \text{, else} \end{cases}$.

Our example clearly respects NoRemoteAccess because only the process state of p_i is represented in a Localview and with the equivalence of the lifted actions in Φ with the non-lifted versions, it follows that a process does not use information of other process states.

Finally we want that events do not change the local state of processes, i.e. events may affect every entry of a configuration but $\mathbf{S}$. Therefore we postulate that for all $\mathfrak{C}, \mathfrak{C}' \in \mathbb{C}$ and all $\mathcal{A} \in \Psi$ the implication

$$\mathcal{A}(\mathfrak{C}, \mathfrak{C}') \Rightarrow \mathbf{S}_{\mathfrak{C}} = \mathbf{S}_{\mathfrak{C}'} \tag{StateInvEv}$$

must hold.

As defined in Figure 2.4 *SendMsg* and *Deliver* do not change process states. Hence our example respects StateInvEv.

Based on the stipulated assertions, we define a Distributed algorithm as follows:

Definition 2.3.2.3 (Distributed Algorithm)
A Distributed Algorithm $\mathfrak{A}$ *is a tuple* $(\ \mathrm{Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\mathrm{ad}},\ \mathrm{C2LV},\ \mathrm{LV2C}\ )$ *where*

1. $(\ \mathrm{Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\mathrm{ad}}\ )$ *is an algorithm*
2. C2LV *and* LV2C *respect locViewProp1 and locViewProp2*
3. *every* $\mathcal{A} \in \Phi$ *respects LocalviewRespect and NoRemoteAccess*
4. *every* $\mathcal{A} \in \Psi$ *respects StateInvEv*

We have already seen that $\mathrm{C2LV}_{\mathrm{flood}}$ and $\mathrm{LV2C}_{\mathrm{flood}}$ respect locViewProp1 and locViewProp2, that every $\mathcal{A} \in \Phi_{\mathrm{flood}}$ respects LocalviewRespect and NoRemoteAccess and every event in Ψ_{flood} respects StateInvEv. Moreover $\mathfrak{A}_{\mathrm{flood}}$ is an algorithm. By Definition 2.3.2.3 it follows that $(\ \mathrm{Init}_{\mathrm{flood}},\ \Phi_{\mathrm{flood}},\ \Psi_{\mathrm{flood}},\ \mathcal{F}_{\mathrm{flood}},\ \mathrm{C2LV}_{\mathrm{flood}},\ \mathrm{LV2C}_{\mathrm{flood}}\ )$ is a Distributed Algorithm.

Note that the difference between our definitions of an Algorithm and a Distributed algorithms is a restriction of modelling concepts. Therefore, many results in this work also hold for the more general concept of an algorithm. Hence, many of the following theorems are shown for algorithms and only when needed we use the concept of a Distributed Algorithm.

We conclude this section by establishing that LocalviewRespect and locViewProp2 assure that no process manipulates the state of other processes in a given Distributed Algorithm:

Lemma 2.3.2.4 (StateInv1)
Let $\mathfrak{A} = (\ \mathrm{Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\mathrm{ad}},\ \mathrm{C2LV},\ \mathrm{LV2C}\ )$ *be a Distributed Algorithm. For all* $p_i, p_j \in \mathbb{P}$*, for all* $\mathfrak{C}, \mathfrak{C}' \in \mathbb{C}$ *and all actions* $\mathcal{A} \in \Phi$ *the following implication holds:*

$$\mathcal{A}(\mathfrak{C}, \mathfrak{C}', p_i) \wedge (\boldsymbol{S}_{\mathfrak{C}}(p_j) \neq \boldsymbol{S}_{\mathfrak{C}'}(p_j)) \Rightarrow (p_i = p_j)$$

Proof. $\hookrightarrow$ Isabelle[3]

2.3.3. Modelling Actions

Considering the definition of a Distributed Algorithm it seems as much effort is needed to model an algorithm and moreover to prove that a modelled algorithm is actually a Distributed Algorithm. This section will explain how this effort can be reduced to a minimum.

Regarding the different tasks, defining the initial states and the Localview mappings C2LV and LV2C is rather simple and fast, because it does not depend on the complexity of the algorithm and can be similar from one algorithm to the next (cf. the case studies in Chapter 5). Even the tasks to show that events do not change process states and therefore satisfy StateInvEv and to show all process-actions respect NoRemoteAccess will usually be more or less obvious (depending on the Modelling of the actions). Most work is required to model the process-actions of an algorithm and to show that these actions satisfy LocalviewRespect. At first glance

[3]Isabelle/HOL theory: `DistributedAlgorithm.thy`, Lemma(s): `StateInv(1)`

LocalviewRespect seems to double the effort needed to model one process-action, because for every modelled action $\mathcal{A}$ a non-lifted version $a^{\downarrow}_{\mathcal{A}}$ has to be given as a witness to show that there is one. Moreover, for each $\mathcal{A}$ we have to show $\mathcal{A} = \text{Lift}(a^{\downarrow}_{\mathcal{A}})$. Of course it is possible to model all non-lifted versions of all actions $a^{\downarrow}_1, \ldots, a^{\downarrow}_n$ and then define $\Phi \triangleq \left\{ \text{Lift}(a^{\downarrow}_1),\ \ldots,\ \text{Lift}(a^{\downarrow}_n) \right\}$. This reduces the effort for Modelling the algorithm and for proving consistency but we do not recommend this strategy because the effort for the later correctness proofs will vastly increase due to the definition of Lift. As an example consider the typical case where we know that some predicate $P : \mathbb{C} \to \mathbf{bool}$ holds for $\mathfrak{C}$ (hence we have $P(\mathfrak{C})$) and we want to show that if there is a step $\mathfrak{C} \to_{p_i:\mathcal{A}} \mathfrak{C}'$ then $P(\mathfrak{C}')$ holds as well. Let us further assume we defined $\mathcal{A} \triangleq \text{Lift}(a^{\downarrow}_{\mathcal{A}})$ where $a^{\downarrow}_{\mathcal{A}}$ is the concrete implementation of the intended process-action. Hence we have to show

$$P(\mathfrak{C}) \wedge \left(\mathfrak{C} \to_{p_i:\text{Lift}(a^{\downarrow}_{\mathcal{A}})} \mathfrak{C}' \right) \Rightarrow P(\mathfrak{C}')$$

which is reduced to

$$P(\mathfrak{C}) \wedge \left(\exists lv' \,.\, a^{\downarrow}_{\mathcal{A}}(\text{C2LV}(\mathfrak{C}, p_i), lv') \wedge \mathfrak{C}' = \text{LV2C}(lv', p_i, \mathfrak{C}) \right) \Rightarrow P(\mathfrak{C}')$$

by unfolding the definition of Lift. Even in a paper proof it would not be easy to deal with the existential quantifier for lv' and to handle it within a theorem prover will clearly be even more difficult. Since this kind of reasoning occurs very often (why this can be regarded as a typical case will be explained in Sections 3.1 and 3.2) this is harmful for most of the cases.

Moreover, while time costs spent for the effort in modelling each action twice and doing the respective equivalence proofs is constant over the number of proofs that have to be done to show correctness, the required reasoning for dealing with Lift scales with the number of invariants that have to be proven. Since it seems still not satisfying to model each process-action twice, we propose an alternative approach. As we will see this approach requires to model each action only once and additionally does rather reduce the effort for proving correctness.

To understand why we need the bounded variable lv' and the respective existential quantifier in the definition of Lift, consider the following alternative definition for a similar function Lift$'$ where we replaced lv' by $\text{C2LV}(\mathfrak{C}', p_i)$. This would be the expected natural option for lv':

$$\begin{aligned} \text{lf}'_{a^{\downarrow}}(\mathfrak{C}, \mathfrak{C}', p_i) &\triangleq a^{\downarrow}(\text{C2LV}(\mathfrak{C}, p_i), \text{C2LV}(\mathfrak{C}', p_i)) \wedge \mathfrak{C}' = \text{LV2C}(\text{C2LV}(\mathfrak{C}', p_i), p_i, \mathfrak{C}) \\ \text{Lift}'(a^{\downarrow}) &\triangleq \text{lf}'_{a^{\downarrow}} \end{aligned} \qquad \text{(AltLiftA)}$$

This definition is indeed very similar to the original and, at first sight, seems to solve the reasoning problems, since there is no existential quantifier anymore. The problem occurs when we try to model blind writes as we did for the outbox in our example. Remember that, in our example, C2LV sets the value of the outbox to $\emptyset$. The Definition AltLiftA requires $a^{\downarrow}(\text{C2LV}(\mathfrak{C}, p_i), \text{C2LV}(\mathfrak{C}', p_i))$ and hence there must be a non-lifted step $a^{\downarrow}(lv, lv')$ where the outbox is still $\emptyset$ in the successor configuration lv', which would imply that it is impossible to send a message. More generally speaking, with this alternative definition we eliminate the option to model blind writes (writes where the process is not allowed to read the current value but is able to affect the value in the successor configuration, see example). Therefore we maintain

Definition 2.3.2.2.

But still it is the required existential quantifier in the definition of Lift that makes it impossible to effectively generate lifted versions from non-lifted by using Lift. Handling this quantifier is difficult because there can be several witnesses for lv' which must be regarded in the proofs (we have to show whenever there is one, the implication holds). In real world applications (and furthermore in many theoretical studies) local process-actions are usually deterministic (even in our case studies). Hence, whenever a non-lifted action $a^{\downarrow}$ is enabled, there is only one option for lv' such that $a^{\downarrow}(lv, lv')$ holds. For a deterministic version of $a^{\downarrow}$ we need only to assert for which $a^{\downarrow}$ is enabled and how to compute the successive Localview lv' from a given Localview lv. Hence, we define:

Definition 2.3.3.1 (Deterministic non-lifted Action (DnA))
A Deterministic non-lifted Action (DnA) is a tuple $d^{\downarrow} = (\ en,\ \Delta\ )$, where

1. *$en : \mathbb{L} \rightarrow \textbf{bool}$ returns true if and only if $d^{\downarrow}$ is enabled.*
2. *$\Delta : \mathbb{L} \rightarrow \mathbb{L}$ is a function which computes the successive Localview lv' for a given Localview lv.*

We write ${d^{\downarrow}}_{en}$ to denote the enabled predicate of a DnA $d^{\downarrow}$ and
${d^{\downarrow}}_{\Delta}$ to denote the second entry,

Let $\mathbb{D}_{nA} \triangleq ((\mathbb{L} \rightarrow \textbf{bool}) \times (\mathbb{L} \rightarrow \mathbb{L}))$ denote the set of all DnAs. Analogously to Lift we define a function DLift : $\mathbb{D}_{nA} \rightarrow (\mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \textbf{bool})$ that returns a lifted version $\mathcal{A}_{d^{\downarrow}}$ for a given DnA $d^{\downarrow}$:

Definition 2.3.3.2 (DLift)
For an arbitrary DnA $d^{\downarrow}$ let $\mathrm{dlf}_{d^{\downarrow}} : \mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \textbf{bool}$ *denote the following function:*

$$\mathrm{dlf}_{d^{\downarrow}}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq {d^{\downarrow}}_{en}(\mathrm{C2LV}(\mathfrak{C}, p_i)) \wedge \mathfrak{C}' = \mathrm{LV2C}({d^{\downarrow}}_{\Delta}(\mathrm{C2LV}(\mathfrak{C}, p_i)), p_i, \mathfrak{C})$$

Then DLift : $\mathbb{D}_{nA} \rightarrow (\mathbb{C} \times \mathbb{C} \times \mathbb{P} \rightarrow \textbf{bool})$ *is defined as:*

$$\mathrm{DLift}(d^{\downarrow}) \triangleq \mathrm{dlf}_{d^{\downarrow}}$$

This implies that we can lift DnAs without the burden of an existential quantifier. Therefore DnAs can effectively be used for (invariant) reasoning. The following theorem abates the proof obligation for lifted DnAs concerning the LocalviewRespect assumption:

Theorem 2.3.3.3
Let $d^{\downarrow} \in \mathbb{D}_{nA}$ be a DnA. There is a non-lifted version (of type $\mathbb{L} \times \mathbb{L} \rightarrow \textbf{bool}$) for DLift($d^{\downarrow}$).

Proof. $\hookrightarrow$ Isabelle[4]

This theorem is very helpful for modelling algorithms and manifests the main advantage of DnAs: It shows that there is a (standard) non-lifted action of type $\mathbb{L} \times \mathbb{L} \rightarrow \textbf{bool}$ for every DnA. This enables us to model (at least all deterministic) process-actions by giving only the

[4]Isabelle/HOL theory: `DistributedAlgorithm.thy`, Lemma(s): `DLiftImpLocalViewRespect1`

corresponding DnAs. Assume that we defined an algorithm using DnAs $d_1^{\downarrow}, \ldots, d_k^{\downarrow}$. Then we can define the set Φ of process-actions as $\Phi \triangleq \left\{ \text{DLift}(d_1^{\downarrow}),\ \ldots,\ \text{DLift}(d_k^{\downarrow}) \right\}$. The proof obligation for equation LocalviewRespect is abated by Theorem 2.3.3.3.

This yields a second advantage: defining lifted actions requires to make assertions over the entries of configurations and thus over global behaviour. A DnA makes assertions over local behaviour. Therefore we need only to define the local transitions which is obviously more adequate to describe local behaviour. For our example we can model the action *SendToAll* by the DnA $\mathit{SendToAll}^{\downarrow} \triangleq (\mathit{STA}_{en}, \mathit{STA}_{\Delta})$ with

$$\begin{aligned} \mathit{STA}_{en}(lv) &\triangleq (\forall m_r \in \mathcal{M} \,.\, \mathbf{In}_{lv}(m_r) = 0) \\ \mathit{STA}_{\Delta}(lv) &\triangleq ((\mathbf{S}_{lv}, \mathbf{In}_{lv}, \mathit{New}_{\text{msgs}}(p_i, \mathbf{S}_{lv}), \mathbf{ID}_{lv})) \end{aligned} \tag{SendToAllDNA}$$

Let us now consider the *ProcessMsg* action of our example. This is a typical case for a non-deterministic action: there is some message m_r such that m_r is processed. Based on a Localview lv we cannot determine a fixed Localview lv' such that we transit from lv to lv' by executing *ProcessMsg* since we do not know which message will be processed. In a real system, perhaps this would be the first message in a buffer or the first message on a stack but this is not part of our model and therefore it is not possible to give a DnA that is equivalent to *ProcessMsg*. Similarly we could also use something like a Choose operator (cf. [Lam02]) but that is not equivalent to the given Definition *ProcessMsg* because it also chooses a value deterministic and the given definition does not.

Therefore for the remaining Non-deterministic actions we have the two already discussed options:

1. Model both versions (lifted and non-lifted), show that they are equivalent and work with the lifted version for reasoning.

2. Model only the non-lifted version and hazard the consequences of the increased effort within the proofs.

Of course for a real system the adequate option would be to choose the best deterministic implementation.

2.3.4. Limitations of the Model

In the last section we showed how we can model variables that processes can access and how to avoid unintended modelling of access to non-shared variables.

Let us now consider an example that depicts the boundaries of our model. In this example (see Figure 2.6) we model an algorithm where the involved processes share (unsynchronised) access on a central storage. This storage stores exactly one natural number. Moreover processes store a natural number in their process state. We will refer to this value by 'p_i's value' for a process p_i. Configurations are therefore vectors $(\mathbf{S}, \mathbf{n})$ where $\mathbf{n} \in \mathbb{N}$ is the value of the central storage and $\mathbf{S} : \mathbb{P} \to \mathbb{N}$ represents the array of process states which is in this example an array of natural numbers. As described for Notation 2.3.1.2 for a configuration $\mathfrak{C} = (\mathbf{S}, \mathbf{n})$, we write $\mathbf{n}_{\mathfrak{C}}$

to denote the entry $\mathbf{n}$ for the storage of $\mathfrak{C}$ and $\mathbf{S}_{\mathfrak{C}}$ to denote the entry for the array of process states in $\mathfrak{C}$. Initially the storage contains a value $\mathbf{n}_0 \in \mathbb{N}$ with $\mathbf{n}_0 < 200$ and the initial values for the process states are determined by an arbitrary function $\mathrm{v_{inp}} : \mathbb{P} \to \mathbb{N}$.

Three atomic process actions are defined to access the storage:

$Read_{\mathrm{store}}$: A process p_i can read the value from the storage. The value will then be written into the own process state (which can be interpreted as a local variable of p_i in its own memory).

$Write_{\mathrm{store}}$: A process p_i can write a value to the storage. In this example p_i adds an arbitrary value $m \in \mathbb{N}$ to its own value and writes the result to the storage and back into memory.

$Reset_{\mathrm{store}}$: If a process p_i recognises that its value is greater than 400, it is able to reset the storage and writes 0 to the storage and back into memory.

$$
\begin{aligned}
\mathrm{Init}_{\mathrm{store}} &\triangleq \{\ (\mathrm{v_{inp}}, n_0)\ \mid\ n_0 \in \mathbb{N} \wedge n_0 < 200\ \} \\
Read_{\mathrm{store}}(\mathfrak{C}, \mathfrak{C}', p_i) &\triangleq \left(\mathfrak{C}' = \left(\left(\lambda p_j . \begin{cases} \mathbf{n}_{\mathfrak{C}} & \text{, if } p_i = p_j \\ \mathbf{S}_{\mathfrak{C}}(p_j) & \text{, else} \end{cases}\right), \mathbf{n}_{\mathfrak{C}}\right)\right) \\
Write_{\mathrm{store}}(\mathfrak{C}, \mathfrak{C}', p_i) &\triangleq \left(\exists m \in \mathbb{N}.\ \mathfrak{C}' = \left(\left(\lambda p_j . \begin{cases} \mathbf{S}_{\mathfrak{C}}(p_i) + m & \text{, if } p_i = p_j \\ \mathbf{S}_{\mathfrak{C}}(p_j) & \text{, else} \end{cases}\right), \mathbf{S}_{\mathfrak{C}}(p_i) + m\right)\right) \\
Reset_{\mathrm{store}}(\mathfrak{C}, \mathfrak{C}', p_i) &\triangleq (\mathbf{S}_{\mathfrak{C}}(p_i) > 400 \\
&\qquad \wedge\ \mathfrak{C}' = \left(\left(\lambda p_j . \begin{cases} 0 & \text{, if } p_i = p_j \\ \mathbf{S}_{\mathfrak{C}}(p_j) & \text{, else} \end{cases}\right), 0\right)) \\
\Phi_{\mathrm{store}} &\triangleq \{\ Write_{\mathrm{store}},\ Read_{\mathrm{store}},\ Reset_{\mathrm{store}}\ \} \\
\Psi_{\mathrm{store}} &\triangleq \emptyset \\
\mathrm{Prio}_{\mathrm{store}}(R) &\triangleq (\forall t \in \mathrm{dom}(R)\ .\ \forall p_i \in \mathbb{P}\ .\ \mathbf{Enabled}_{A}(Reset, R(t), p_i) \\
&\qquad \Rightarrow (\forall \mathcal{A} \in \{\ Write_{\mathrm{store}},\ Read_{\mathrm{store}}\ \}\ .\ \neg A(R(t), R(t+1), p_i))) \\
\mathcal{F}_{\mathrm{store}} &\triangleq \{\ \mathrm{Prio}_{\mathrm{store}}\ \} \\
\mathfrak{A}_{\mathrm{store}} &\triangleq (\ \mathrm{Init}_{\mathrm{store}},\ \Phi_{\mathrm{store}},\ \Psi_{\mathrm{store}},\ \mathcal{F}_{\mathrm{store}}\ )
\end{aligned}
$$

Figure 2.6.: Algorithm with Shared Storage

In this scenario we define no events. A predicate $\mathrm{Prio}_{\mathrm{store}}$ is used to assert that if $Reset_{\mathrm{store}}$ is enabled this action is prioritised over all other actions. From an algorithmic point of view this algorithm shows typical interference problems as for example the value of the storage can be reset more than once although it has only violated the bound of 400 for one time. The reason is that we choose the (probably) right abstraction for the $Write_{\mathrm{store}}$ and $Reset_{\mathrm{store}}$ actions where a process p_i is not able to read and write the storage in one atomic step. Otherwise we rather would have modelled $Reset_{\text{store}}$ such that it uses the original value from the storage and not p_i's value:

$$Reset^{\text{omnis}}_{\text{store}}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq \left(\mathbf{n}_{\mathfrak{C}} > 400 \wedge \mathfrak{C}' = \left(\left(\lambda p_j. \begin{cases} 0 & \text{, if } p_i = p_j \\ \mathbf{S}_{\mathfrak{C}}(p_j) & \text{, else} \end{cases}\right), \mathbf{n}_{\mathfrak{C}}\right)\right)$$

Hence the latter 'omniscient' definition of $Reset^{\text{omni}}_{\text{store}}$ would imply that a process p_i always sees the current value of the storage. Obviously in the given example this is not intended and hence it is modelled as described by Figure 2.6. But regarding the example this is only implicitly given by the chosen implementation. Of course we would like to use the means introduced in Sections 2.3.2 and 2.3.3 to model the read and write options of a process, but since the same functions C2LV and LV2C are used for all actions, we would have to decide if a process p_i can either see the current value of $\mathbf{n}$ or has to do blind writes. If we choose to model the blind write (which would be appropriate for the $Write_{\text{store}}$ and the $Reset_{\text{store}}$ actions) we are not able to read the value within the $Read_{\text{store}}$ action. Hence, we have to allow read and write access to $\mathbf{n}$. What we suggest for such a situation is to provide two variables in a Localview: one to which p_i is only allowed to write and which is initially undefined and one which p_i is allowed to read. This does not prevent unintended read access but makes the distinction between the read and writes to $\mathbf{n}$ more obvious. Furthermore, Section 2.3.7 will show how we provide a more elaborate model for shared memory that provides subactions to prevent wrong access to a shared storage.

2.3.5. Interprocess Communication (IPC): Message Passing

An important part of a distributed system is the communication infrastructure the system is based on. Without communication there is no distributed system but only a collection of autonomous processes. Different options of communication mechanisms are possible. A common communication model is message passing. Here processes are able to send and receive point-to-point messages. Point-to-point communication means that there is a reserved channel for each pair of processes in the system [Fuz08]. If there is an upper bound on communication delays, then we speak of *synchronous* message passing. Since we deal with problems that are only interesting in asynchronous models, we use a model of asynchronous message passing, i.e. a model where no bound for the delay of messages is known. Moreover we assume that there is no guarantee that messages are delivered in order (i.e. a reordering of the messages is possible). As in our example in Figure 2.4, we assume that processes can put multiple point-to-point messages to their outbox at once but as long as the messages are in the outbox message loss by process crash is still possible. Hence, there is no guarantee that if one of the messages reaches the receiver all other messages will because if the sender crashes after only one messages of many has been transmitted to the media, messages left in the outbox will be lost. Moreover, for some applications message loss during transmission is part of the model. Therefore, we use two different models for message passing based on the assumed reliability:

Quasi-reliable Message passing: There is a guarantee that all messages sent from a correct sender to a correct receiver will eventually be delivered. Note that in this context a process is *correct* if it does not crash in the given run. A more general definition for *correct* will be given later.

Message passing with message loss: Messages might get lost during the transmission although sender and receiver do not crash due to lossy channels.

We also provide means to model a duplication of messages. Hence we will regard both mis-use cases: the loss of a message and a duplication of a message. The fundamental concepts for our model of message passing are already introduced in our example in 2.4 and are mainly adopted from [Fuz08].

Quasi-reliable Message Passing

Certainly, the basic entity for our message passing model is a message. A message m is a n-tuple $m = (snd, rcv, cnt, \ldots)$ where $snd, rcv \in \mathbb{P}$ are sender and receiver of the message and cnt is a content of an arbitrary type α (note that in our example in Figure 2.4 $\alpha = \mathbb{N}$) represents the content of the message. Of course a message can be extended by further information in the $\ldots$ part of the message, e.g. a local time stamps for send or receive data. We denote the sender of a message m by snd_m, the receiver of m by rcv_m, and the content of m by cnt_m. In our model a message traverses three states on its way from the sender to the receiver (cf. [Fuz08] and [KNR12]):

- **outgoing**: When a sender wants to send a message (or a set of messages) it puts the message into its outgoing buffer. Messages in the outgoing buffer are still at the senders site, i.e. outgoing messages are lost if the sender crashes.

- **transit**: The message is on its way to the receiver.
  A crash of the sender does no longer concern messages that are in **transit**, but if the receiver crashes the message will always be in **transit**.

- **received**: The message has already arrived on the receiver's site.
  It is now ready to be processed by the receiver.

These states are referred to as *message tags* and we define (cf. Section 2.3.2)

$$\mathrm{QPP}_{\mathcal{T}} \triangleq \{\ \textbf{outgoing},\ \textbf{transit},\ \textbf{received}\ \}\,.$$

As for the example in Figure 2.4, we use a multiset-like structure to represent messages in our system, i.e. for every message the number of copies that are **outgoing**, (respectively **transit**, **received**) are stored by mapping pairs of messages and message tags $(\ m,\ t\ )$ to the number of copies that exist of m with status t. For such a mapping we use the term *Message History* (in our example the Message History is represented by the $\mathbf{M}$ part of the configuration). The Message History is part of each configuration of algorithms that use message passing and represents the state of the message evolution. Analogously to [Fuz08] we use $\mathbf{Q}$ to denote the Message History of an algorithm with quasi-reliable message passing. Therefore the standard configuration with quasi-reliable message passing is of the form:

$$\begin{pmatrix} \mathbf{S} \\ \vdots \\ \mathbf{Q} \\ \vdots \end{pmatrix}$$

As always, we refer to the array of process states of a configuration $\mathfrak{C}$ by $\mathbf{S}_{\mathfrak{C}}$ and to the Message History of $\mathfrak{C}$ by $\mathbf{Q}_{\mathfrak{C}}$.

To work with the messages in a Message History, we define the following message sets based on the set $\mathcal{M}$ which is the set of all messages:

Definition 2.3.5.1 (Message Sets)
Let $\boldsymbol{Q} : \mathcal{M} \times \mathrm{QPP}_{\mathcal{T}} \rightarrow \mathbb{N}$ be a Message History.
*Then we define the set of all **outgoing** messages for **Q** as:*

$$\mathrm{outMsgs}_{\,\boldsymbol{Q}} \triangleq \{\, m \in \mathcal{M} \;\mid\; \boldsymbol{Q}(m, \boldsymbol{outgoing}) > 0 \,\},$$

*the set of all messages in **transit** as:*

$$\mathrm{transitmsgs}_{\,\boldsymbol{Q}} \triangleq \{\, m \in \mathcal{M} \;\mid\; \boldsymbol{Q}(m, \boldsymbol{transit}) > 0 \,\}$$

*and the set of all **received** messages as:*

$$\mathrm{recmsgs}_{\,\boldsymbol{Q}} \triangleq \{\, m \in \mathcal{M} \;\mid\; \boldsymbol{Q}(m, \boldsymbol{received}) > 0 \,\}$$

*Finally, we define the set of existing messages in a Message History **Q** as:*

$$\mathrm{msgs}_{\,\boldsymbol{Q}} \triangleq \mathrm{outMsgs}_{\,\boldsymbol{Q}} \cup \mathrm{transitmsgs}_{\,\boldsymbol{Q}} \cup \mathrm{recmsgs}_{\,\boldsymbol{Q}}$$

For Message Histories we provide subactions that can be used within actions to describe the transition from one Message History $\mathbf{Q}_{\mathfrak{C}}$ to the Message History $\mathbf{Q}_{\mathfrak{C}'}$ in the successive configuration $\mathfrak{C}'$.

At first, we define the Message History $\mathrm{sndupd}_{\mathbf{Q},M} : \mathcal{M} \times \mathrm{QPP}_{\mathcal{T}} \rightarrow \mathbb{N}$, which is the update of Message History $\mathbf{Q}$ resulting from sending a given set of messages $M \subseteq \mathcal{M}$ in $\mathbf{Q}$:

$$\mathrm{sndupd}_{\mathbf{Q},M}(m,t) \triangleq \begin{cases} \mathbf{Q}(m,t) + 1 & ,\text{if } (t = \mathbf{outgoing} \wedge m \in M) \\ \mathbf{Q}(m,t) & ,\text{else} \end{cases}$$

Using sndupd we can now define the subaction

$$\mathrm{MultiSend} : (\mathcal{M} \times \mathrm{QPP}_{\mathcal{T}} \rightarrow \mathbb{N}) \times (\mathcal{M} \times \mathrm{QPP}_{\mathcal{T}} \rightarrow \mathbb{N}) \times \mathcal{P}(\mathcal{M}) \rightarrow \mathbf{bool}$$

to model the sending of a set of messages $M \subseteq \mathcal{M}$:

$$\mathrm{MultiSend}(\mathbf{Q}, \mathbf{Q}', M) \triangleq \left(\mathbf{Q}' = \mathrm{sndupd}_{\mathbf{Q},M}\right)$$

Using function update we define subactions for sending a single message m, transmitting a message m (setting its status from **outgoing** to **transit**) and for receiving a message m (setting its status from **transit** to **received**) in a step from a Message History $\mathbf{Q}$ to $\mathbf{Q}'$:

$$\begin{aligned}
\text{Send}(\mathbf{Q},\mathbf{Q}',m) &\triangleq \mathbf{Q}(\ m,\ \textbf{outgoing}\ ) := \mathbf{Q}(m,\textbf{outgoing}) + 1\\
\text{Transmit}(\mathbf{Q},\mathbf{Q}',m) &\triangleq m \in \text{outMsgs}_{\mathbf{Q}} \wedge (\mathbf{Q}(\ m,\ \textbf{outgoing}\ ) := \mathbf{Q}(m,\textbf{outgoing}) - 1)\\
&\qquad\qquad\qquad\qquad [(m,\textbf{transit}) := \mathbf{Q}(m,\textbf{transit}) + 1]\\
\text{Receive}(\mathbf{Q},\mathbf{Q}',m) &\triangleq m \in \text{transitmsgs}_{\mathbf{Q}} \wedge (\mathbf{Q}(\ m,\ \textbf{transit}\ ) := \mathbf{Q}(m,\textbf{transit}) - 1)\\
&\qquad\qquad\qquad\qquad [(m,\textbf{received}) := \mathbf{Q}(m,\textbf{received}) + 1]
\end{aligned}$$

Finally, we define a subaction to describe the duplication of a message m, which is modelled by simply incrementing the **transit** number of copies of m:

$$\text{Duplicate}(\mathbf{Q},\mathbf{Q}',m) \triangleq m \in \text{transitmsgs}_{\mathbf{Q}} \wedge (\mathbf{Q}(\ m,\ \textbf{transit}\ ) := \mathbf{Q}(m,\textbf{transit}) + 1)$$

As an example for the application of our introduced definitions for Message History consider the rewritten algorithm $\mathfrak{A}_{\text{flood}}$ (cf. Section 2.3.2) in Figure 2.7.

$$\begin{aligned}
\textit{ProcessMsg}(\mathfrak{C},\mathfrak{C}',p_i) &\triangleq \exists m_r \in \text{recmsgs}_{\mathbf{Q}_{\mathfrak{C}}}\ .\ \text{rcv}_{m_r} = p_i\\
&\quad \wedge \mathbf{M}_{\mathfrak{C}'} = \mathbf{M}_{\mathfrak{C}}[(m_r,\textbf{received}) := \mathbf{M}_{\mathfrak{C}}(m_r,\textbf{received}) - 1]\\
&\quad \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{\mathfrak{C}}(p_i) + 1]\\
\textit{SendToAll}(\mathfrak{C},\mathfrak{C}',p_i) &\triangleq \{\ m \in \text{recmsgs}_{\mathbf{Q}_{\mathfrak{C}}}\ \mid\ \text{rcv}_m = p_i\ \} = \emptyset\\
&\quad \wedge \text{MultiSend}(\mathbf{Q}_{\mathfrak{C}},\mathbf{Q}_{\mathfrak{C}'},\textit{New}_{\text{msgs}}(p_i,\mathbf{S}_{\mathfrak{C}}(p_i)))\\
&\quad \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}}\\
\textit{SendMsg}(\mathfrak{C},\mathfrak{C}') &\triangleq \exists m_o\ .\ \text{Send}(\mathbf{Q}_{\mathfrak{C}},\mathbf{Q}_{\mathfrak{C}'},m_0)\\
\textit{Deliver}(\mathfrak{C},\mathfrak{C}') &\triangleq \exists m_t\ .\ \text{Receive}(\mathbf{Q}_{\mathfrak{C}},\mathbf{Q}_{\mathfrak{C}'},m_0)
\end{aligned}$$

Figure 2.7.: Simple Algorithm $\mathfrak{A}_{\text{flood}}$ rewritten with Message Definitions

Obviously the definitions help to write more compact specifications. For the work with DnAs the definitions help to define LV2C. As an example consider the following alternative definition for LV2C$_{\text{flood}}$ (cf. the old definition of LV2C$_{\text{flood}}$ in Section 2.3.3):

$$\hat{\mathbf{M}}^{p_i}_{lv',\mathfrak{C}}(m,t) \triangleq \begin{cases} \min(\mathbf{In}_{lv'}(m),\mathbf{M}_{\mathfrak{C}}(m,\textbf{received})) & ,\text{if } t = \textbf{received}\\ & \quad\wedge\ \text{rcv}_m = p_i\\ \text{sndupd}_{\mathbf{M}_{\mathfrak{C}},\mathbf{Out}_{lv'}}(m,t) & ,\text{else}\end{cases}$$

$$\text{LV2C}_{\text{flood}}(lv', p_i, \mathfrak{C}) \triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{lv}] \\ \hat{\mathbf{M}}^{p_i}_{lv',\mathfrak{C}} \end{pmatrix} \qquad (\text{LV2C}_{\text{flood}})$$

Some proofs require assertions about how many messages of each status (**outgoing**, **transit**, **received**) are in the system. We assume that there are no messages in the message history at system start and only finitely many messages are added in every step to the Message History **Q**. Then we define the following functions to determine the number of messages for each status:

Definition 2.3.5.2
*Let **Q** be a Message History.*

*The number of **outgoing** messages can be determined by:*

$$\text{SumOutgoing}_{\boldsymbol{Q}} \triangleq \sum_{m \in \text{msgs}_{\boldsymbol{Q}}} \boldsymbol{Q}(m, \boldsymbol{outgoing})$$

*The number of **transit** messages can be determined by:*

$$\text{SumTransit}_{\boldsymbol{Q}} \triangleq \sum_{m \in \text{msgs}_{\boldsymbol{Q}}} \boldsymbol{Q}(m, \boldsymbol{transit})$$

*The number of **received** messages can be determined by:*

$$\text{SumReceived}_{\boldsymbol{Q}} \triangleq \sum_{m \in \text{msgs}_{\boldsymbol{Q}}} \boldsymbol{Q}(m, \boldsymbol{received})$$

*Finally the sum of all messages in **Q** is:*

$$\text{SumMsgs}_{\boldsymbol{Q}} \triangleq \sum_{m \in \text{msgs}_{\boldsymbol{Q}}} (\text{SumOutgoing}_{\boldsymbol{Q}} + \text{SumTransit}_{\boldsymbol{Q}} + \text{SumReceived}_{\boldsymbol{Q}})$$

Of course the definition for $\text{SumOutgoing}_{\mathbf{Q}}$ is equivalent to

$$\text{SumOutgoing}_{\mathbf{Q}} = \sum_{m \in \text{recmsgs}_{\mathbf{Q}}} \mathbf{Q}(m, \mathbf{outgoing})$$

and respectively:

$$\text{SumTransit}_{\mathbf{Q}} = \sum_{m \in \text{transitmsgs}_{\mathbf{Q}}} \mathbf{Q}(m, \mathbf{transit})$$
$$\text{SumReceived}_{\mathbf{Q}} = \sum_{m \in \text{recmsgs}_{\mathbf{Q}}} \mathbf{Q}(m, \mathbf{received}).$$

Using these definitions we can determine how many transitions due to the movement from messages in $\text{msgs}_{\mathbf{Q}}$ one status to the next are possible. Because every message that is outgoing can do two moves (to the **transit** status and then to **received**) remaining message moves are calculated by:

$$\mathrm{MaxRemMsgMoves}_{\mathbf{Q}} \triangleq 2 \cdot \mathrm{SumOutgoing}_{\mathbf{Q}} + \mathrm{SumTransit}_{\mathbf{Q}}$$

Of course this value increases if new messages are added to the system.
The value $\mathrm{MaxRemMsgMoves}_{\mathbf{Q}}$ can be helpful to find out how many steps the system can make before all messages are processed if it is already known that no further messages will be sent.

By the given definitions the following basic properties hold for all Message Histories $\mathbf{Q}$:

Proposition 2.3.5.3 (Message Subsets)
Let $\boldsymbol{Q}$ be a Message History.

1. $\mathrm{outMsgs}_{\boldsymbol{Q}} \subseteq \mathrm{msgs}_{\boldsymbol{Q}}$
2. $\mathrm{transitmsgs}_{\boldsymbol{Q}} \subseteq \mathrm{msgs}_{\boldsymbol{Q}}$
3. $\mathrm{recmsgs}_{\boldsymbol{Q}} \subseteq \mathrm{msgs}_{\boldsymbol{Q}}$

Proof. $\hookrightarrow$ Isabelle[5]

If we have a MultiSend step from a Message History $\mathbf{Q}$ to $\mathbf{Q}'$ then the messages in M will be added to the **outgoing** messages and hence:

Proposition 2.3.5.4 (MultiSend **Properties)**
If there is a MultiSend *step (formally:* $\mathrm{MultiSend}(\boldsymbol{Q}, \boldsymbol{Q}', M)$*) for two Message Histories $\boldsymbol{Q}$ and $\boldsymbol{Q}'$ and a set of messages M then*

1. $\mathrm{outMsgs}_{\boldsymbol{Q}'} = \mathrm{outMsgs}_{\boldsymbol{Q}} \cup M$
2. $\mathrm{transitmsgs}_{\boldsymbol{Q}'} = \mathrm{transitmsgs}_{\boldsymbol{Q}}$
3. $\mathrm{recmsgs}_{\boldsymbol{Q}'} = \mathrm{recmsgs}_{\boldsymbol{Q}}$
4. $\mathrm{msgs}_{\boldsymbol{Q}} \subseteq \mathrm{msgs}_{\boldsymbol{Q}'}$
5. $\mathrm{msgs}_{\boldsymbol{Q}'} = \mathrm{msgs}_{\boldsymbol{Q}} \cup M$

Proof. $\hookrightarrow$ Isabelle[6]

For the subactions Send, Transmit, Receive we can make respective propositions, which are provided in appendix A (Propositions A.0.0.2, A.0.0.3, A.0.0.4).

It is easy to see that for a Transmit and Receive step the set msgs does not change:

Proposition 2.3.5.5 (msgs **Invariant)**
If there is a Transmit *or* Receive *step for two Message Histories $\boldsymbol{Q}$ and $\boldsymbol{Q}'$ and a message $m \in \mathcal{M}$ (formally:* $\mathrm{Transmit}(\boldsymbol{Q}, \boldsymbol{Q}', m) \vee \mathrm{Receive}(\boldsymbol{Q}, \boldsymbol{Q}', m)$*) then*

$$\mathrm{msgs}_{\boldsymbol{Q}} = \mathrm{msgs}_{\boldsymbol{Q}'}$$

[5]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `MsgsProps`
[6]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `MultiSendProps,MultiSendMsgChange`

Proof. $\hookrightarrow$ Isabelle[7]

Furthermore it is strictly determined how a Transmit step affects SumOutgoing, SumTransit and SumReceived:

Lemma 2.3.5.6 (Transmit **Invariant)**
If there is a Transmit *step for two Message Histories* $\boldsymbol{Q}$ *and* $\boldsymbol{Q'}$ *and a message* m *(formally:* $\text{Transmit}(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) then*

1. $\text{SumOutgoing}_{\boldsymbol{Q}} > 0$
2. $\text{SumOutgoing}_{\boldsymbol{Q'}} = \text{SumOutgoing}_{\boldsymbol{Q}} - 1$
3. $\text{SumTransit}_{\boldsymbol{Q'}} = \text{SumTransit}_{\boldsymbol{Q}} + 1$
4. $\text{SumReceived}_{\boldsymbol{Q'}} = \text{SumReceived}_{\boldsymbol{Q}}$

Proof. $\hookrightarrow$ Isabelle[8]

The respective lemma for Receive is:

Lemma 2.3.5.7 (Receive **Invariant)**
If there is a Receive *step for two Message Histories* $\boldsymbol{Q}$ *and* $\boldsymbol{Q'}$ *and a message* m *(formally:* $\text{Receive}(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) then*

1. $\text{SumTransit}_{\boldsymbol{Q}} > 0$
2. $\text{SumOutgoing}_{\boldsymbol{Q'}} = \text{SumOutgoing}_{\boldsymbol{Q}}$
3. $\text{SumTransit}_{\boldsymbol{Q'}} = \text{SumTransit}_{\boldsymbol{Q}} - 1$
4. $\text{SumReceived}_{\boldsymbol{Q'}} = \text{SumReceived}_{\boldsymbol{Q}} + 1$

Proof. $\hookrightarrow$ Isabelle[9]

Respective lemmas for Send, MultiSend and Duplicate trivially hold. By Lemma 2.3.5.6 and 2.3.5.7 we can deduce that the sum of messages SumMsgs is not affected by Transmit and Receive steps:

Theorem 2.3.5.8
If there is a Transmit *or* Receive *step for two Message Histories* $\boldsymbol{Q}$ *and* $\boldsymbol{Q'}$ *and a message* m *(formally:* $\text{Transmit}(\boldsymbol{Q}, \boldsymbol{Q'}, m) \vee \text{Receive}(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) then*

$$\text{SumMsgs}_{\boldsymbol{Q'}} = \text{SumMsgs}_{\boldsymbol{Q}}$$

Proof. $\hookrightarrow$ Isabelle[10]

Finally, we define a property that characterises what it means for the point-to-point message passing to be *quasi-reliable* in our context:

[7]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `MsgsInvariant`
[8]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `TransmitInv`
[9]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `ReceiveInv`
[10]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `SumInv`

Definition 2.3.5.9 (QPPReliability)
Let R be a sequence of configurations and let $\text{Correct}(R)$ *denote the set of correct processes in R.*
R satisfies the property of QPPReliability *if and only if for all times $t > 0$ and for all messages $m \notin \text{msgs}(R(t-1))$ holds:*

$$m \in \text{outMsgs}_{R(t)} \wedge \text{rcv}_m \in \text{Correct}(R) \wedge \text{snd}_m \in \text{Correct}(R) \Rightarrow \big(\exists u > t \,.\, m \in \text{recmsgs}_{R(u)}\big)$$

Note that we assume there are no messages in the system at time $t = 0$. The QPPReliability property can for example be added to the set $\mathcal{F}_{\text{ad}}$ of a Distributed Algorithm
$\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}},\ \text{C2LV},\ \text{LV2C })$ if the message infrastructure of $\mathfrak{A}$ is assumed to be quasi reliable. An example for this is the algorithm in Section 5.2.

These are the fundamentals for our model of quasi-reliable message passing. The next section will explain the extensions that are required to model message loss.

Message Passing with Message Loss

For some algorithms we want to prove that they solve distributed problems even in the case of lossy channels, i.e. in environments where messages might get lost on their way from sender to receiver. Therefore we propose an option to extend our model of Message Histories by the notion of *message loss*. At first we introduce a new status **lost** for messages. Hence we define:

$$\hat{\mathcal{T}}_{\text{ags}} \triangleq \{\ \textbf{outgoing},\ \textbf{transit},\ \textbf{received},\ \textbf{lost}\ \}$$

A Message History with message loss is then a mapping $\mathbf{C} : \mathcal{M} \times \hat{\mathcal{T}}_{\text{ags}} \to \mathbb{N}$ (note that we use $\mathbf{Q}$ for quasi-reliable Message Histories but $\mathbf{C}$ for Message Histories, where messages can also be lost). Hence Message Histories with message loss work as the quasi-reliable Message Histories but additionally provide a fourth status such that it is possible for a message to get **lost**. Once a message has the status **lost**, it will never be received. The status **lost** is only reachable for a message that currently has status **transit**.

Beside the already introduced sets $\text{outMsgs}_{\mathbf{C}}$, $\text{transitmsgs}_{\mathbf{C}}$, and $\text{recmsgs}_{\mathbf{C}}$ we define the set of lost messages for a Message History $\mathbf{C}$:

$$\text{lostmsgs}_{\mathbf{C}} \triangleq \{\ m \in \mathcal{M} \ \mid\ \mathbf{C}(m, \textbf{lost}) > 0\ \}$$

For a Message History with message loss we have:

$$\text{msgs}_{\mathbf{C}} \triangleq \text{outMsgs}_{\mathbf{C}} \cup \text{transitmsgs}_{\mathbf{C}} \cup \text{recmsgs}_{\mathbf{C}} \cup \text{lostmsgs}_{\mathbf{C}}$$

And beside the definitions of Send, Transmit, Receive, and Duplicate we define a subaction $\text{Lose} : (\mathcal{M} \times \hat{\mathcal{T}}_{\text{ags}} \to \mathbb{N}) \times (\mathcal{M} \times \hat{\mathcal{T}}_{\text{ags}} \times \text{msgs} \to \mathbb{N}) \to \textbf{bool}$:

$$\begin{aligned}\text{Lose}(\mathbf{C}, \mathbf{C}', m) \triangleq\ & m \in \text{transitmsgs}_{\mathbf{C}} \wedge \\ & \mathbf{C}' = (\mathbf{C}[(m, \textbf{transit}) := \mathbf{C}(m, \textbf{transit}) - 1])\,[(m, \textbf{lost}) := \mathbf{C}(m, \textbf{lost}) + 1]\end{aligned}$$

We extend Proposition 2.3.5.3 for our notion of message loss as follows:

Proposition 2.3.5.10 (Message Subsets)
Let $\boldsymbol{C}$ be a Message History.

1. $\text{outMsgs}_{\,\boldsymbol{C}} \subseteq \text{msgs}_{\,\boldsymbol{C}}$
2. $\text{transitmsgs}_{\,\boldsymbol{C}} \subseteq \text{msgs}_{\,\boldsymbol{C}}$
3. $\text{recmsgs}_{\,\boldsymbol{C}} \subseteq \text{msgs}_{\,\boldsymbol{C}}$
4. $\text{lostmsgs}_{\,\boldsymbol{C}} \subseteq \text{msgs}_{\,\boldsymbol{C}}$

Proof. $\hookrightarrow$ Isabelle[11]

And we extend the MultiSend-, Send-,Transmit- and Receive-Properties propositions by

- $\text{lostmsgs}_{\,\mathbf{C'}} = \text{lostmsgs}_{\,\mathbf{C}}$

Finally we assert how the Lose subaction affects the sets outMsgs, transitmsgs, recmsgs, lostmsgs and msgs:

Proposition 2.3.5.11 (Lose **Properties)**
If there is a Lose *step (formally:* $\text{Lose}(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) for two Message Histories* $\boldsymbol{C}$ *and* $\boldsymbol{C'}$ *and a message* m *then*

1. $\text{outMsgs}_{\,\boldsymbol{Q'}} = \text{outMsgs}_{\,\boldsymbol{Q}}$
2. $\text{transitmsgs}_{\,\boldsymbol{Q'}} \subseteq \text{transitmsgs}_{\,\boldsymbol{Q}}$
3. $\text{recmsgs}_{\,\boldsymbol{Q'}} = \text{recmsgs}_{\,\boldsymbol{Q}}$
4. $\text{lostmsgs}_{\,\boldsymbol{Q'}} = \text{lostmsgs}_{\,\boldsymbol{Q}} \cup \{\ m\ \}$
5. $\text{msgs}_{\,\boldsymbol{Q}} \subseteq \text{msgs}_{\,\boldsymbol{Q'}}$
6. $\text{msgs}_{\,\boldsymbol{Q'}} = \text{msgs}_{\,\boldsymbol{Q}} \cup\ M$

Proof. $\hookrightarrow$ Isabelle[12]

2.3.6. Interprocess Communication (IPC): Broadcasts

Of course, in a multiparticipant system, there is not only a need for bilateral relationships but also for interactions between more than two peers. Therefore the broadcast abstraction is introduced to enable communication between multiple processes: every single process is able to send a message not only to one process but to a group of processes at once by using the broadcast mechanism.

[11]Isabelle/HOL theory: `MsgPassWLoss.thy`, Lemma(s): `MsgsProps`
[12]Isabelle/HOL theory: `MsgPassWLoss.thy`, Lemma(s): `LoseProps`

Distinguished by their guarantees [CGR11] introduces several broadcast abstractions:

- Best-effort broadcast
- (Regular) Reliable Broadcast
- Uniform Reliable Broadcast
- Stubborn broadcast
- Probabilistic Broadcast
- Causal Broadcast

The *Best-effort* Broadcast abstraction gives the following guarantees ([CGR11]):

Validity: If a correct process broadcasts a message m, then every correct process eventually delivers m.

No duplication: No message is delivered more than once.

No creation: If a process delivers a message m with sender s, then m was previously broadcasted by process s.

In this work we focus on the (regular) Reliable Broadcast abstraction, which will be used for two algorithms of our case study (see Section 5.2 and Section 5.3).

The *Reliable Broadcast* abstraction adds an *Agreement* property to the guarantees of Best-effort Broadcasts and uses a slightly weaker *Validity* property (cf. [CGR11], [Fuz08]):

Validity: If a correct process p broadcasts a message m, then p eventually delivers m.

No Duplication: No message is delivered more than once.

No Creation: If a process delivers a message m with sender s, then m was previously broadcasted by process s.

Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every process.

We adopt the ideas from [Fuz08] for our formal model of a Reliable Broadcast mechanism and introduce an entry $\mathbf{B}$, which keeps track of the broadcasts ($\mathbf{B}$ is the *Broadcast History*). We assume that broadcasts are unique (every message is broadcasted at most once) and model $\mathbf{B}$ as a set to which broadcasts are added as pairs $bc = (cnt, P)$ where cnt is the content of broadcast bc and P is the subset of processes that has not yet delivered bc. Every broadcast initially is added by the sender with $P = \mathbb{P}$ (we assume that a broadcast is always sent to all processes) and a process that delivers a broadcast removes itself from P. Hence if $\mathbf{B}_{\mathfrak{C}} = \{ \ (\ cnt, \ \{ \ p_1, \ p_2 \ \} \ ) \ \}$ then in configuration $\mathfrak{C}$ all processes but p_1 and p_2 already delivered the broadcast with content cnt.

Based on this notion, we define a subaction Rbc_Broadcast to send a broadcast with content *cnt* and a subaction Rbc_Deliver that can be used by a process p_i to deliver broadcasts with content *cnt* within a step from Broadcast History $\mathbf{B}$ to $\mathbf{B}'$:

$$\begin{aligned}
\text{Rbc\_Broadcast}(\mathbf{B}, \mathbf{B}', \mathit{cnt}) &\triangleq (\mathbf{B}' = (\mathbf{B} \cup \{\ (\ \mathit{cnt},\ \mathbb{P}\ )\ \}))\\
\text{Rbc\_Deliver}(\mathbf{B}, \mathbf{B}', \mathit{cnt}, p_r) &\triangleq \exists bc \in \mathbf{B}\ .\ \exists P\ .\ bc = (\ \mathit{cnt},\ P\ ) \wedge p_r \in P\\
&\quad \wedge \mathbf{B}' = (\mathbf{B} \setminus \{\ bc\ \}) \cup \{\ (\ \mathit{cnt},\ P \setminus \{\ p_r\ \}\ )\ \}
\end{aligned}$$

Note that Rbc_Broadcast does not require the sender p_s as an argument since the sender is not part of the broadcast message and $p_s \in \mathbb{P}$ is automatically added to set of processes that have to deliver the broadcast. In contrast for the delivery of a broadcast the process p_r that executes Rbc_Deliver is needed as an argument, since it must be removed from the set of processes that still have to deliver the broadcast in $\mathbf{B}'$. Examples for the application of broadcasts and the use of these subactions are given in Sections 5.2 and 5.3.

2.3.7. Interprocess Communication (IPC): Shared Memory

As a third option for interprocess communication we introduce the notion of *shared memory* where multiple processes can access the same memory area. Hence, for processes it is possible to communicate by writing to a dedicated shared area of memory, which other processes are supposed to read afterwards.

Literature distinguishes different kinds of shared memory dependent on the conditions that are assumed for the access on the different parts of the shared memory. In a seminal paper Lamport [Lam85] introduces the following shared memory abstractions as forms of persistent communication:

- Safe Registers
- Regular Registers
- Atomic Registers

All three abstractions assume that a register can be read by multiple reader processes but there is only one dedicated process that is allowed to write values to the register, i.e. for every process p_i there is exactly one register Reg_{p_i} to which p_i is allowed to write. Read and write operations to the registers are assumed to be non-atomic, i.e. read and write operations may overlap in time. Therefore there might be interference between overlapping (i.e. *concurrent*) reads and writes. Generally it is assumed that read operations do not affect other read operations and hence there is no interference between two reads. Since every process p_i performs actions sequentially and is only allowed to write to its own register Reg_{p_i}, concurrent writes to the same registers are ruled out. Therefore interference can only occur between a single write and one or more reads. For a *Safe Register* there is no assumption about the value that is returned by a read operation that is concurrent to a write but only that it is one of the possible values of the register [Lam85]. Note that a read operation that is not concurrent to a write returns the most recent value that was written to the register (respectively the initial value if there is no such value). Lamport furthermore defines that ‘a *Regular Register* is a register, which is safe and in which a read that overlaps a write obtains either the old or new value’ [Lam85]. As explained before, *safe* means here that a read not concurrent with a write gets the correct value. The following example (depicted in Figure 2.8) illustrates the difference between a *Regular Register* and an *Atomic Register*: assume two consecutive read operations Rd_1 and Rd_2 to the same register R where Rd_2 starts strictly after Rd_1 has finished. Assume furthermore that both reads are concurrent to the same write operation Wr, which overwrites an old value v by a new value v'. Assuming an Atomic Register the result of the two read operations behaves as one would normally expect, i.e.

- if Rd_1 returns v' then Rd_2 must return v'
- if Rd_1 returns v then Rd_2 returns either v or v'.

In contrast for a Regular Register we only assume Rd_1 and Rd_2 return either v or v' and hence the case where Rd_1 returns the newer value v' while Rd_2 still returns the old value v is not ruled out.

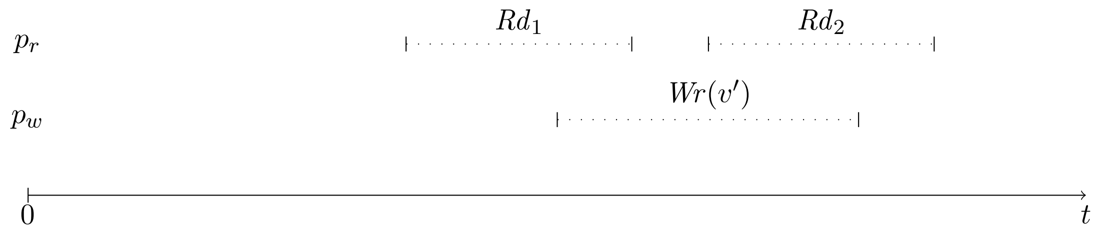

Figure 2.8.: Concurrent read and write operations

In the following we explain how we formally model Regular Registers and describe how this model can be adapted for Safe and Atomic Registers.

A Formal Model for Regular Registers

The difficulty in modelling Lamports register abstractions lies in the kind of specification of these abstractions. By the definition of runs, a run in our model is a sequence of configurations. Hence there is no explicit notion of actions for a run, i.e. given two succeeding configurations $R(t)$ and $R(t+1)$ in a run R, it might be impossible to deduce which action caused the step from $R(t)$ to $R(t+1)$ because more than one action may be able to cause the same result. This means that our model is state-based, but obviously the definitions and assumptions for the register abstractions (see previous subsection) are action based (based on the notions of reads and writes). This implies that for the representations of Regular Registers in our model, we need to develop a state based formal model of the Regular Register abstraction and prove that our model is equivalent to the action-based specification.

We use a module **Sh** for our Regular Register abstraction and call it a *Shared Memory History*. At first let us consider the view a fixed process p_i has on the shared memory. We section p_i's view into two parts:

- The (write-oriented) view p_i has on the register it is allowed to write.
- The (read-oriented) view on all registers.

Let us give an example for this: As explained, we assume that there is a register Reg_{p_i} for each process $p_i \in \mathbb{P}$. Let us further assume p_i writes a value v' in its register Reg_{p_i} where the value v is stored before v' replaces v. The following considerations explains how this is reflected by the write-oriented view of p_i and the read-oriented views of all processes on p_i's register. Firstly, we distinguish between stable periods of the register (between write operations) and transient periods (during the write operations). In the write-oriented view of p_i, there is an old value v in the register and v is supposed to be replaced by v'. Thus, during a transient period of writing, in p_i's view there are two values v and v' assigned to Reg_{p_i}. For the stable periods, where there is no write to Reg_{p_i}, p_i knows that Reg_{p_i} contains the value p_i has written most recently and therefore there is only one value v for stable periods. In our example this is v before the write operation starts and v' after the write operation is finished. Hence for write-oriented views we have a value v for the stable periods but a pair of values (v, v') for the transient periods.

In the read-oriented view we store the values for each register Reg_{p_j}, that could be currently returned if p_i reads Reg_{p_j} based on the last begin of a read operation to Reg_{p_j}. As an example assume again that there is a process p_w, which overwrites a value v in Reg_{p_i} with a value v'. During the write operation of p_w another process p_r starts reading Reg_{p_w}. Of course when p_r starts reading, both values, v and v', are possible candidates for the return of the read operation. While p_r has not finished reading Reg_{p_w} every write operation of p_w adds further values to the set of possible return values. Hence if p_w finishes writing v' and starts writing a value v'' before p_r has finished reading Reg_{p_w}, the set of possible returns is (at least) $\{\ v,\ v',\ v''\ \}$. This is our notion of the read-oriented view on the registers.

Let $\mathcal{V}_{\mathrm{reg}}$ denote the set of values that can be stored in a register. We then define the view of a process p_i on the status of the registers (for short: *p_i's register status value*) as a tuple $\mathrm{rst}_{p_i} = (\ rPhs,\ \mathrm{view}^{\mathrm{w}},\ \mathrm{view}^{\mathrm{r}}\ )$ where *rPhs* is the phase of p_i concerning read and write operations of registers. Since p_i is supposed to be a sequential process p_i's phase can be either reading a register Reg_{p_j} (denoted by $\mathbf{rReading}_{p_j}$) or writing its own register (denoted by **rWriting**) or neither reading nor writing (denoted by **rIdle**). We denote the set of possible register phases as:

$$\mathcal{R}_{phs} \triangleq \{\ \mathbf{rReading}_{p_j}, \mathbf{rWriting}, \mathbf{rIdle}\ \mid\ p_j \in \mathbb{P}\ \}.$$

$\mathrm{view}^{\mathrm{w}} \in (\mathcal{V}_{\mathrm{reg}} \cup (\mathcal{V}_{\mathrm{reg}} \times \mathcal{V}_{\mathrm{reg}}))$ is the write-oriented view on the state of the register of p_i and finally $\mathrm{view}^{\mathrm{r}} : \mathbb{P} \to \mathcal{P}(\mathcal{V}_{\mathrm{reg}})$ is the read-oriented view of p_i on all registers. Hence $\mathrm{view}^{\mathrm{w}}$ is either a value $v \in \mathcal{V}_{\mathrm{reg}}$ (if the register is stable) or a pair $(v, v') \in (\mathcal{V}_{\mathrm{reg}} \times \mathcal{V}_{\mathrm{reg}})$ where v is the old and v' the new value during a transient periods of a write operation. $\mathrm{view}^{\mathrm{r}}(p_j)$ is p_i's read-oriented view on p_j's Register Reg_{p_j} and therefore a set $V \subseteq \mathcal{V}_{\mathrm{reg}}$ of register values that are possible candidates for a read return if p_i would finish the read of Reg_{p_j} with the current read-view.

By $\mathfrak{R}$ we denote the set of all register status values. We define the module $\mathbf{Sh} : \mathbb{P} \to \mathfrak{R}$ as a mapping over processes such that $\mathbf{Sh}(p_i)$ returns the register status $(\ rPhs,\ \mathrm{view}^{\mathrm{w}},\ \mathrm{view}^{\mathrm{r}}\ )$ for a process $p_i \in \mathbb{P}$. As always for a configuration $\mathfrak{C} = \big(\mathbf{S}, \ldots, \mathbf{Sh}, \ldots\big)$ we use the notation $\mathbf{Sh}_{\mathfrak{C}}$ to denote the **Sh** entry of $\mathfrak{C}$. Moreover we use $rPhs \triangleright \mathbf{Sh}(p_i)$ to denote the register phase for process $p_i \in \mathbb{P}$ in a Shared Memory History **Sh**, $\mathrm{view}^{\mathrm{w}} \triangleright \mathbf{Sh}(p_i)$ to denote the write-oriented view of p_i in $\mathbf{Sh}$ and $\text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)(p_j)$ to denote p_i's read-oriented view on the register of p_j in $\mathbf{Sh}$ (cf. Notation 2.3.1.2).

WriteBegin - Now we can define a subaction for the begin of a write (*WriteBegin*) that happens in a step from a Shared Memory History $\mathbf{Sh}$ to a shared Memory History $\mathbf{Sh'}$ where p_i wants to write value v' to its register. Consider the following definitions (an explanation is given below):

$$\text{viewupdr}^{v'}_{\mathbf{Sh},p_i}(p_j) \triangleq (\text{view}^{\text{r}} \triangleright \mathbf{Sh}\,(p_j))\,[p_i := (\text{view}^{\text{r}} \triangleright \mathbf{Sh}\,(p_j))\,(p_i) \cup \{\ v'\ \}]$$

$$\text{rwbupd}^{(v,v')}_{\mathbf{Sh},p_i}(p_j) \triangleq \begin{cases} \left(\mathit{rPhs} \triangleright \mathbf{Sh}(p_j), \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_j), \text{viewupdr}^{v'}_{\mathbf{Sh},p_i} \right) & \text{, if } p_i \neq p_j \\ \left(\mathbf{rWriting}, (v, v'), \text{viewupdr}^{v'}_{\mathbf{Sh},p_i} \right) & \text{, if } p_i = p_j \end{cases}$$

$$\begin{aligned} \mathit{WriteBegin}(\mathbf{Sh}, \mathbf{Sh'}, p_i, v') \triangleq\ & \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rIdle} \\ & \wedge \exists v \in \mathcal{V}_{\text{reg}}\ .\ \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i) = v \wedge \mathbf{Sh'} = \text{rwbupd}^{(v,v')}_{\mathbf{Sh},p_i} \end{aligned}$$

If p_i begins to write a value v' in a step from a Shared Memory History $\mathbf{Sh}$ to the next Shared Memory History $\mathbf{Sh'}$ then:

- the register phase of p_i is **rIdle**
- there is a value v such that
  - v is the old value in the write-oriented view of p_i's register
  - all read-oriented views of all processes p_j on p_i's register are updated by adding v' to the possible return values (this is done by $\text{viewupdr}^{v'}_{\mathbf{Sh},p_i}(p_j)$)
  - the new register phase of p_i in $\mathbf{Sh'}$ is **rWriting** (cf. $\text{rwbupd}^{(v,v')}_{\mathbf{Sh},p_i}(p_j)$)
  - (v, v') is the new value in the write-oriented view of p_i (cf. $\text{rwbupd}^{(v,v')}_{\mathbf{Sh},p_i}(p_j)$)
- everything else is unchanged from $\mathbf{Sh}$ to $\mathbf{Sh'}$

WriteEnd - If p_i finishes a write, in the current Shared Memory History $\mathbf{Sh}$ there must be a pair of register values $(\ v,\ v'\ ) \in (\mathcal{V}_{\text{reg}} \times \mathcal{V}_{\text{reg}})$ such that $(\ v,\ v'\ )$ is the value in the write-oriented view of p_i and the register phase of p_i. In the subsequent Shared Memory History the value in the write-oriented view of p_i must be set to v', and the register phase has to be set to **rIdle**. We define a respective subaction *WriteEnd*:

$$\begin{aligned} \mathit{WriteEnd}(\mathbf{Sh}, \mathbf{Sh'}, p_i) \triangleq\ & \exists v, v' \in \mathcal{V}_{\text{reg}}\ .\ \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i) = (v, v') \\ & \wedge \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rWriting} \\ & \wedge \mathbf{Sh'} = \mathbf{Sh}[p_i := (\mathbf{rIdle}, v', \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i))] \end{aligned}$$

We define the following function $\text{vset} : \mathcal{V}_{\text{reg}} \cup (\mathcal{V}_{\text{reg}} \times \mathcal{V}_{\text{reg}}) \rightarrow \mathcal{P}\,(\mathcal{V}_{\text{reg}})$ to convert the values of a write-oriented view to a set, by putting either two values $v, v' \in \mathcal{V}_{\text{reg}}$ of a pair or one value

$v \in \mathcal{V}_{\text{reg}}$ into a set:

$$\text{vset}(vx) = \begin{cases} \{\ v,\ v'\ \} & ,\text{if } vx = (\ v,\ v'\ ) \\ \{\ vx\ \} & ,\text{else} \end{cases}$$

RdBegin - To start reading a register Reg_{p_j} in a Message History $\mathbf{Sh}$, a process p_i has to put the values from the write-oriented view of p_j ($\text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_j)$) to its own read-oriented view on the register of p_j. Furthermore the register phase of p_i has to be **rIdle** in **Sh** and in the next Message History $\mathbf{Sh'}$ the register phase must be $\mathbf{rReading}_{p_j}$ and the read-oriented view is set to ($\text{vset}\,(\text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_j))$):

$$\begin{aligned} \mathit{RdBegin}(\mathbf{Sh}, \mathbf{Sh'}, p_i, p_j) \triangleq\ & \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rIdle} \\ & \wedge \mathbf{Sh'} = \mathbf{Sh}[p_i := (\mathbf{rReading}_{p_j}, \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i), \\ & \qquad \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)[p_j := \text{vset}\,(\text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_j))])] \end{aligned}$$

RdEnd - Finally we define a subaction *RdEnd* to be used to finish a read operation. $\mathit{RdEnd}(\mathbf{Sh}, \mathbf{Sh'}, p_i, p_j, v)$ is true if p_i finishes a read operation for register Reg_{p_j} by a transition from **Sh** to $\mathbf{Sh'}$ with reading the value $v \in \mathcal{V}_{\text{reg}}$. This must be a value from p_i's read-oriented view on register Reg_{p_j}. The transition from **Sh** to $\mathbf{Sh'}$ must be set the register phase of p_i from $\mathbf{rReading}_{p_j}$ to **rIdle**.

$$\begin{aligned} \mathit{RdEnd}(\mathbf{Sh}, \mathbf{Sh'}, p_i, p_j, v) \triangleq\ & \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rReading}_{p_j} \\ & \wedge v \in \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)(p_j) \\ & \wedge \mathbf{Sh'} = \mathbf{Sh} \\ & \qquad [i := (\ \mathbf{rIdle},\ \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i),\ \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)\ )] \end{aligned}$$

The usage of this subaction should be $\exists v \in \mathcal{V}_{\text{reg}}\ .\ \mathit{RdEnd}(\mathbf{Sh}, \mathbf{Sh'}, p_i, p_j, v)$, i.e. there must be no restrictions for v. Otherwise we would allow only to read predetermined values from the register.

Let us consider the example for the read- and write-oriented views in Figure 2.9. We have four processes p_w, p_{r1}, p_{r2} and p_{r3}. We assume that p_{r1}, p_{r2} and p_{r3} read the register Reg_{p_w}. p_{r1} and p_{r3} perform two reads each, p_{r2} performs only one read and p_w performs three consecutive write operations. Our system starts at time t_0. At this time the write-oriented view of p_w ($\text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_w)$) is set to an initial value i_{p_w}. At time t_1 process p_w begins writing the value 5 to its register by executing *WriteBegin* (WB) with value 5. During the transient phase from t_1 until t_2 the write-oriented view of p_w is (i_{p_w}, 5). At time t_2 p_w is finished with the first write and calls *WriteEnd* (WE). Hence the write-oriented view of p_w during the subsequent stable phase is set to 5. At time t_3 the processes p_{r1}, p_{r2} and p_{r3} start reading the register Reg_{p_w} by executing *RdBegin* (RB). Therefore their read-oriented view ($\text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_{r1})(p_w), \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_{r2})(p_w)$ and $\text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_{r3})(p_w)$) contains only the value of the write-oriented view of p_w: 5. At time t_4 the first read operation of p_{r1} is finished. Since p_{r1}'s read-oriented view on p_w's register contains only one value, the only option for the return value of the read is 5. At time t_5 p_w starts writing

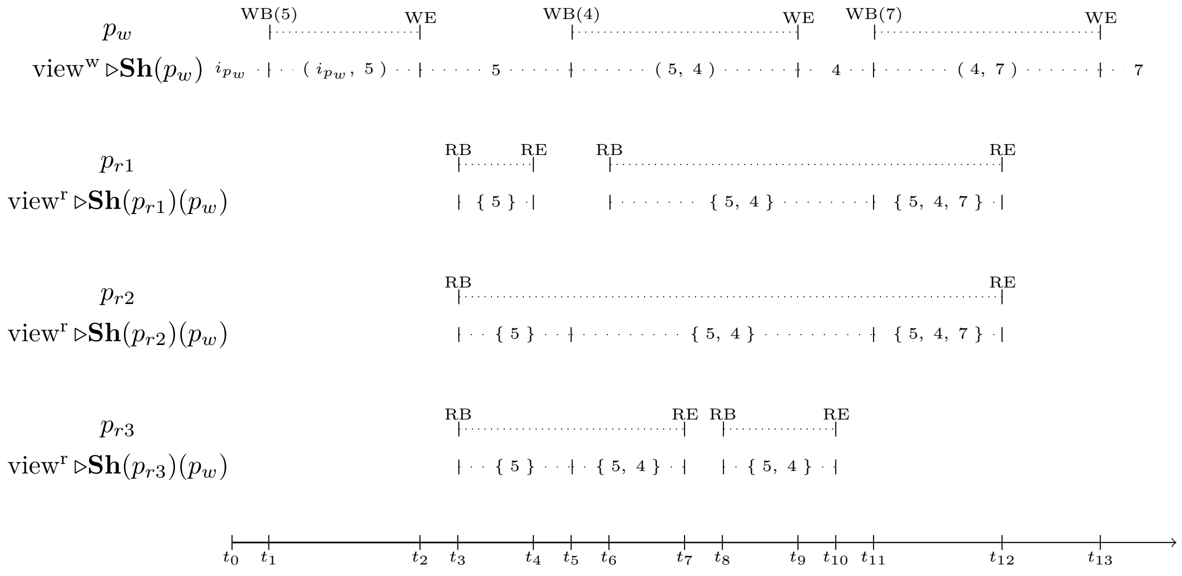

Figure 2.9.: Read- and write-oriented views

again. This time p_w writes the value 4. Therefore p_w's write-oriented view is set to (5, 4). Since p_{r2} and p_{r3} have not yet finished reading, value 4 is added to the sets that represent the read-oriented view of p_{r2} and p_{r3}. Before p_w finishes writing, at time t_6 p_{r1} starts reading again. p_w is therefore still in a transient period at t_6 and therefore both values of the pair (5, 4) are added to the read-oriented view of p_{r1}. At t_7 process p_{r3} finishes its read operation while its read-oriented view is the set { 5, 4 }. Therefore *RdEnd* in t_7 must either return 5 or 4. At t_8 process p_{r3} starts reading again and since p_w is still in a transient period 5 and 4 are again applied to read-oriented view of p_{r3}. At t_9 p_w finishes writing by executing *WriteEnd*. Shortly after, at time t_{10} process p_{r3} finishes reading for the second time. Again the read-oriented view of p_{r3} is { 5, 4 } and therefore the result is again either 5 or 4. Therefore here we have the case were the first read of p_{r3} might return the new value (4) but the second possibly returns the old value (5) although the first read is finished before the second read started. At t_{11} the writer p_w starts writing again, this time for the value 7. Hence both read-oriented views are updated by adding the value 7 to the respective sets. Hence the read-oriented views of p_{r1} and p_{r2} are set to { 5, 4, 7 }. Therefore values returned by the ending reads in t_{12} are chosen from the set { 5, 4, 7 }. Finally p_w finishes writing at t_{13}.

Note that the right sequential arrangement of begin and end operations are enforced by the register phase of the processes (at least as long as we access the shared memory by the defined subactions). If we start with the register phase **rIdle** we are allowed to execute *RdBegin* or alternatively *WriteBegin*, but we cannot execute *RdEnd* and *WriteEnd* because these subactions require the phase to be either $\mathbf{rReading}_{p_j}$ or **rWriting**. Then after executing *RdBegin* (respectively *WriteBegin*) we enter $\mathbf{rReading}_{p_j}$ (**rWriting**) and now we are able to execute *RdEnd* (*WriteEnd*) and therefore to return to phase **rIdle**. The possible transitions between the register phases are depicted in Figure 2.10. Let σ_{sh} = { *WriteBegin*, *WriteEnd*, *RdBegin*, *RdEnd* } be

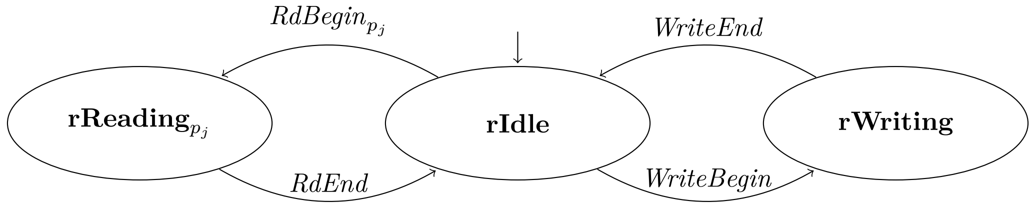

Figure 2.10.: Register phases

the set of the defined subactions. We introduce the notion of a *ProperMemStep* to guarantee that read and write operations are in accordance with these subactions:

Definition 2.3.7.1 (ProperMemStep)
A step $\mathfrak{C} \to \mathfrak{C}'$ is called a ProperMemStep of process p_i
(written formally: ProperMemStep$(\mathfrak{C}, \mathfrak{C}', p_i)$) if and only if

1. $\exists v \in \mathcal{V}_{\text{reg}}$. *WriteBegin*$(\boldsymbol{Sh}_{\mathfrak{C}}, \boldsymbol{Sh}_{\mathfrak{C}'}, p_i, v)$ *or*

2. *WriteEnd*$(\boldsymbol{Sh}_{\mathfrak{C}}, \boldsymbol{Sh}_{\mathfrak{C}'}, p_i)$ *or*

3. $\exists p_j \in \mathbb{P}$. *RdBegin*$(\boldsymbol{Sh}_{\mathfrak{C}}, \boldsymbol{Sh}_{\mathfrak{C}'}, p_i, p_j)$ *or*

4. $\exists p_j \in \mathbb{P}$. $\exists v \in \mathcal{V}_{\text{reg}}$. *RdEnd*$(\boldsymbol{Sh}_{\mathfrak{C}}, \boldsymbol{Sh}_{\mathfrak{C}'}, p_i, p_j, v)$

To ensure the controlled access to the shared memory, we require the following additional properties for a distributed algorithm (Init, Φ, Ψ, $\mathcal{F}_{\text{ad}}$, C2LV, LV2C) that uses the Regular Register Module:

1. There is a mapping initval : $\mathbb{P} \to \mathcal{V}_{\text{reg}}$ from processes to register values such that initval(p_j) returns the initial value of the register Reg_{p_j}.

2. For all processes the initial register phase is **rIdle**.

3. If there is a process-step $\mathfrak{C} \to_{p_i:\mathcal{A}} \mathfrak{C}'$ for a process-action $\mathcal{A} \in \Phi$ by a process $p_i \in \mathbb{P}$ and $\mathbf{Sh}_{\mathfrak{C}} \neq \mathbf{Sh}_{\mathfrak{C}'}$ then there must be a ProperMemStep from $\mathbf{Sh}_{\mathfrak{C}}$ to $\mathbf{Sh}_{\mathfrak{C}'}$ by p_i.

4. If there is an event-step $\mathfrak{C} \to_{\text{ev}:\mathcal{A}} \mathfrak{C}'$ for an event $\mathcal{A} \in \Psi$ then $\mathbf{Sh}_{\mathfrak{C}} = \mathbf{Sh}_{\mathfrak{C}'}$.

We define

$$\begin{aligned} \mathit{InitReg}(\mathbf{Sh}) \triangleq\ & \forall p_i \in \mathbb{P} \,.\, \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rIdle} \\ & \wedge \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i) = \text{initval}(p_i) \\ & \wedge (\forall p_j \in \mathbb{P} \,.\, \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)(p_j) = \emptyset) \end{aligned}$$

and require formally for a Distributed Algorithm (Init, Φ, Ψ, $\mathcal{F}_{\mathrm{ad}}$, C2LV, LV2C) that uses the module for Regular Registers for all configurations $\mathfrak{C}, \mathfrak{C}' \in \mathbb{C}$, for all $\mathcal{A} \in \Phi, \mathcal{A}' \in \Psi$ and for all $p_i \in \mathbb{P}$ the following assertions hold:

$$\mathfrak{C} \in \mathrm{Init} \Rightarrow \mathit{InitReg}(\mathbf{Sh}_{\mathfrak{C}}) \tag{InitRegPred}$$

$$(\mathfrak{C} \rightarrow_{p_i:\mathcal{A}} \mathfrak{C}') \Rightarrow \mathrm{ProperMemStep}(\mathfrak{C}, \mathfrak{C}', p_i) \tag{ProperSteps1}$$

$$(\mathfrak{C} \rightarrow_{\mathrm{ev}:\mathcal{A}'} \mathfrak{C}') \Rightarrow \mathbf{Sh}_{\mathfrak{C}} = \mathbf{Sh}_{\mathfrak{C}'} \tag{ProperSteps2}$$

Equivalence to Lamport's Regular Registers

To see that our model is equivalent to the action-based definition of a Regular Register given by Lamport [Lam85], let us at first consider which values a read of a register are candidates for a return value respecting Lamport's definitions. Let us assume we have a process p_i, which performs a read operation Rd on register Reg_{p_j}, starting at time t_1 and ending at time t_2 in a run R. If there is no concurrent write to Reg_{p_j} we know that the value that is returned has to be the value that was written most recently to Reg_{p_j} by process p_j or $\mathrm{initval}(p_j)$ if there is no such value. Considering our write definitions the former is the value in p_j's write-oriented view during the stable period that follows p_j's last $\mathit{WriteEnd}$ operation before t_1. To determine this value we define a function $\mathit{WrEndsUntil}_{t'} : \Sigma \times \mathbb{P} \rightarrow \mathcal{P}(\mathbb{T})$. For a given run R and a given process p_j this function returns all times until t at which p_j ended a write, i.e. returns the set of all points in time W where for all $t \in W$ process p_j executed a $\mathit{WriteEnd}$ operation at time t in R and $t < t'$. Note that this set is always finite:

$$\mathit{WrEndsUntil}_{t'}(R, p_j) \triangleq \{\ t \in \mathbb{T}\ \mid\ t < t' \wedge \mathit{WriteEnd}(R(t), R(t+1), p_j)\ \}$$

Based on this definition we define a function $\mathit{LastWrEnd}_{t'} : \Sigma \times \mathbb{P} \rightarrow \mathbb{T} \cup \{\ \bot\ \}$, which for a run R and a process p_j returns the most recent time at which p_j executed a $\mathit{WriteEnd}$ operation:

$$\mathit{LastWrEnd}_{t'}(R, p_j) \triangleq \begin{cases} \mathrm{Max}\,(\ \mathit{WrEndsUntil}_{t'}(R, p_j)) & \text{, if } \mathit{WrEndsUntil}_{t'}(R, p_j) \neq \emptyset \\ \bot & \text{, else} \end{cases}$$

Now we are able to define a function $\mathit{LastStableVal}_t : \Sigma \times \mathbb{P} \rightarrow \mathcal{V}_{\mathrm{reg}}$ that returns the last written value if there is one or the initial value for register Reg_{p_j} if the given process p_j has not finished any writes before t':

$$\mathit{LastStableVal}_{t'}(R, p_j) \triangleq \begin{cases} \mathrm{view}^{\mathrm{w}} \triangleright \mathbf{Sh}_{R(\mathit{LastWrEnd}_{t'}(R,p_j)+1)}(p_j) & \text{, if } \mathit{LastWrEnd}_{t'}(R, p_j) \neq \bot \\ \mathrm{initval}(p_j) & \text{, else} \end{cases} \tag{LastStableValue}$$

Considering our read operation Rd, which starts at t_1 and ends at t_2, the value of $\mathit{LastStableVal}_{t_1}(R, p_j)$ is obviously the value Lamport refers to as 'the old value' of the register. Regarding Lamport's definition, this value is the value that must be returned if Rd is not concurrent to a write operation of p_j. Furthermore if we regard the case where there is a write operation concurrent to Rd then this value is still a candidate for the return value of the read operation. The further candidates are then the values Lamport refers to as 'the new values', i.e. all values v for which there exists a concurrent write operation *Wr* such that *Wr* starts writing v to register Reg_{p_j}. Let us consider such an operation *Wr* that writes a value v. Let us assume *Wr* starts with a *WriteBegin* at t_{w1} and may end with a *WriteEnd* at t_{w2}. Note that due to our assumptions InitRegPred, ProperSteps1 and ProperSteps2 there has to be a matching *WriteBegin* for every *WriteEnd*, but there can be a *WriteBegin* without a matching *WriteEnd* (note that this enables us to model a situation where a process crashes before finishing a write operation). For this case we assume $t_{w2} = \infty$. Since *Rd* and *Wr* are concurrent both operations must overlap, i.e. one of the following assertions must hold:

1. *Wr* starts before t_1 but does not end before t_1. Hence: $t_{w1} < t_1$ and $t_{w2} > t_1$ (ConcAssertions)
2. or *Wr* starts during the execution of *Rd*: $t_{w1} > t_1$ and $t_{w1} < t_2$

Note that due to our interleaving model we do not need to consider the case where $t_{w1} = t_1$. For the first case (1.) at time t_1 the value of p_j's write-oriented view at time t_1 must be the pair consisting of the old value $\mathit{LastStableVal}_{t_1}(R, p_j)$ and the new value v (the value *Wr* writes to register Reg_{p_j}). Hence $\text{view}^{\text{w}} \triangleright \mathbf{Sh}_{R(t_1)}(p_j) = (\ \mathit{LastStableVal}_{t_1}(R,\ p_j),\ v\ )$. For the case where there is no concurrent write we already noticed that $\text{view}^{\text{w}} \triangleright \mathbf{Sh}_{R(t_1)}(p_j) = \mathit{LastStableVal}_{t_1}(R, p_j)$. Therefore, obviously $\text{vset}(\text{view}^{\text{w}} \triangleright \mathbf{Sh}_{R(t_1)}(p_j))$ returns the set of possible candidates for the return value of *Rd* for the case where there is no concurrent write access (in this case $\text{vset}(\text{view}^{\text{w}} \triangleright \mathbf{Sh}_{R(t_1)}(p_j)) = \{\ \mathit{LastStableVal}_{t_1}(R,\ p_j)\ \}$) and for the case where there is a concurrent write operation that starts before *Rd* (in this case $\text{vset}(\text{view}^{\text{w}} \triangleright \mathbf{Sh}_{R(t_1)}(p_j)) = \{\ \mathit{LastStableVal}_{t_1}(R,\ p_j),\ v\ \}$). For the second case (2.) where a write operation *Wr* starts writing v during the execution of *Rd*, we know that there is a $\mathit{WriteBegin}(\mathbf{Sh}_{R(t_{w1})}, \mathbf{Sh}_{R(t_{w1}+1)}, p_j, v)$. Putting all together we claim that the set of possible candidates V_{Rd} for the return value of read *Rd* is:

$$\begin{aligned} V_{Rd} = &\big(\text{vset}(\text{view}^{\text{w}} \triangleright \mathbf{Sh}_{R(t_1)}(p_j))\big) \\ &\cup \{\ v \in \mathcal{V}_{\text{reg}} \ \mid\ \exists t' > t_1\ .\ t' < t_2 \wedge \mathit{WriteBegin}(\mathbf{Sh}_{R(t')}, \mathbf{Sh}_{R(t'+1)}, p_j, v)\ \} \end{aligned}$$

Remember that the term $\big(\text{vset}\big(\text{view}^{\text{w}} \triangleright \mathbf{Sh}_{R(t_1)}\,(p_j)\big)\big)$ describes the set of values consisting of the old value of Reg_{p_j} and (if there is such a value) the value a concurrent write operation that has started before *Rd* writes the register Reg_{p_j}. The set $\{\ v \in \mathcal{V}_{\text{reg}}\ \mid\ \exists t' > t_1\ .\ t' < t_2 \wedge \mathit{WriteBegin}(\mathbf{Sh}_{R(t')}, \mathbf{Sh}_{R(t'+1)}, p_j, v)\ \}$ consists of all values, that are written by write operations that started during the execution of *Rd*. As stated before there are no further concurrent write operations and therefore this equation is equivalent to Lamports characterisation of a Regular Register. It remains to show in our model that this set is exactly the set of values the subaction *RdEnd* picks values from. By definition of *RdEnd* an execution $\mathit{RdEnd}(\mathbf{Sh}, \mathbf{Sh}', p_i, p_j, v')$ picks values from p_i's read-oriented view on the register Reg_{p_j}. To show that our model indeed implements a Regular Register, we prove that during a read *Rd* where p_i reads from register Reg_{p_j} the values in p_i's read-oriented view equal the set of possible candidates for a return V_{Rd}:

Theorem 2.3.7.2 (Values in Read-oriented View)
Let R be a run of a Distributed Algorithm $\mathfrak{A} = (\text{ Init}, \; \Phi, \; \Psi, \; \mathcal{F}_{\text{ad}}, \; \text{C2LV}, \; \text{LV2C })$ that respects InitRegPred, ProperSteps1 and ProperSteps1. Let Rd be a read operation that starts reading a register Reg_{p_j} at time t_1 with RdBegin$(\boldsymbol{Sh}_{R(t_1)}, \boldsymbol{Sh}_{R(t_1+1)}, p_i, p_j)$ and ends at time $t_2 \geq t$. Then p_i's read-oriented view on Reg_{p_j} at time t is the following set of values:

$$\begin{aligned}\text{view}^{\text{r}} \rhd \boldsymbol{Sh}_{R(t)}(p_i)(p_j) = \big(&\text{vset}(\text{view}^{\text{w}} \rhd \boldsymbol{Sh}_{R(t_1)}(p_j))\big) \\ &\cup \{ \; v \in \mathcal{V}_{\text{reg}} \; \mid \; \exists t' > t_1 \; . \; t' < t \wedge \mathit{WriteBegin}(\boldsymbol{Sh}_{R(t')}, \boldsymbol{Sh}_{R(t'+1)}, p_j, v) \; \}\end{aligned}$$

Proof. $\hookrightarrow$ Isabelle[13]

Using this result it is easy to show that if there is a read operation *Rd* on a register Reg_{p_j} and there may be concurrent write operations by p_j then the returned value is either one of the concurrently written values or the last stable value of register Reg_{p_j}:

Theorem 2.3.7.3 (Read Result 1)
Let R be a run of a Distributed Algorithm $\mathfrak{A} = (\text{ Init}, \; \Phi, \; \Psi, \; \mathcal{F}_{\text{ad}}, \; \text{C2LV}, \; \text{LV2C })$ that respects InitRegPred, ProperSteps1 and ProperSteps1. Let Rd be a read operation that starts reading a register p_j at time t_1 with RdBegin$(\boldsymbol{Sh}_{R(t_1)}, \boldsymbol{Sh}_{R(t_1+1)}, p_i, p_j)$ and ends at time $t_2 > t_1$ with RdEnd$(\boldsymbol{Sh}_{R(t_2)}, \boldsymbol{Sh}_{R(t_2+1)}, p_i, p_j, v)$. Then at least one of the following assertions holds:

- *$v = \mathit{LastStableVal}_{t_1}(R, p_j)$ or*
- *there is a time t' such that WriteBegin$(\boldsymbol{Sh}_{R(t')}, \boldsymbol{Sh}_{R(t'+1)}, p_j, v)$ and either $t' > t_1 \wedge t' < t_2$ or $t' < t_1$ and the write operation ends either after t_1 or never.*

Proof. $\hookrightarrow$ Isabelle[14]

Note that assertions for t' correspond to the assertions made at ConcAssertions. For a non-concurrent read we show that a read must return the last stable value:

Theorem 2.3.7.4 (Read Result 2)
Let R be a run of a Distributed Algorithm $\mathfrak{A} = (\text{ Init}, \; \Phi, \; \Psi, \; \mathcal{F}_{\text{ad}}, \; \text{C2LV}, \; \text{LV2C })$ that respects InitRegPred, ProperSteps1 and ProperSteps1. Let Rd be a read operation that starts reading a register p_j at time t_1 with RdBegin$(\boldsymbol{Sh}_{R(t_1)}, \boldsymbol{Sh}_{R(t_1+1)}, p_i, p_j)$ and ends at time $t_2 > t_1$ with RdEnd$(\boldsymbol{Sh}_{R(t_2)}, \boldsymbol{Sh}_{R(t_2+1)}, p_i, p_j, v)$.
If there are no concurrent write operations for Reg_{p_j} then

$$v = \mathit{LastStableVal}_{t_1}(R, p_j)$$

Proof. $\hookrightarrow$ Isabelle[15]

Now we are able to establish the equivalence of our and Lamport's model. By Theorems 2.3.7.3 and 2.3.7.4 we can deduce that every value that is read by *RdEnd* in our model is a value, which is a possible candidate for a return by a corresponding read in Lamport's model ($\Rightarrow$).

[13]Isabelle/HOL theory: `Register.thy`, Lemma(s): `ReadResultHelp`
[14]Isabelle/HOL theory: `Register.thy`, theorem(s): `ReadResult1`
[15]Isabelle/HOL theory: `Register.thy`, theorem(s): `ReadResult2`

Also using Theorem 2.3.7.2 we show that if we consider a prefix of a run R where a read operation *Rd* started by a *RdBegin* at t_1 and ended at t_2 for every $v \in V_{Rd}$ there is a continuation of R such that a *RdEnd* returns v($\hookrightarrow$ Isabelle[16]). This implies that every value that is returned by a read operation in Lamport's model is a possible candidate for a return by a corresponding read in our model ($\Leftarrow$).

By ($\Rightarrow$) and ($\Leftarrow$) we get that both models are equivalent.

Safe and Atomic Registers

For modelling a Safe register *RdBegin* can be used without change and for the further subactions we use the following subactions A_{sf}
instead of A for $A \in \{$ *WriteBegin*, *WriteEnd*, *RdEnd* $\}$ (we assume $\perp \notin \mathcal{V}_{\text{reg}}$):

$$\begin{aligned} \mathit{WriteBegin}_{\mathit{sf}}(\mathbf{Sh}, \mathbf{Sh}', p_i, v') \triangleq\ & \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rIdle} \\ & \wedge \mathbf{Sh}' = \text{rwbupd}^{(v',\perp)}_{\mathbf{Sh},p_i} \end{aligned}$$

$$\begin{aligned} \mathit{WriteEnd}_{\mathit{sf}}(\mathbf{Sh}, \mathbf{Sh}', p_i) \triangleq\ & \exists v' \in \mathcal{V}_{\text{reg}} \,.\, \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i) = (v', \perp) \\ & \wedge \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rWriting} \\ & \wedge \mathbf{Sh}' = \mathbf{Sh}[i := (\mathbf{rIdle}, v', \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i))] \end{aligned}$$

$$\begin{aligned} \mathit{RdEnd}_{\mathit{sf}}(\mathbf{Sh}, \mathbf{Sh}', p_i, p_j, v) \triangleq\ & \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rReading}_{p_j} \\ & \wedge (\perp \notin \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)(p_j) \Rightarrow v \in \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)(p_j)) \\ & \wedge \mathbf{Sh}' = \mathbf{Sh} \\ & \qquad [i := (\ \mathbf{rIdle},\ \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i),\ \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)\ )] \end{aligned}$$

These modified subactions provide a $\perp$ in every read-oriented view at the end of every read operation that has been concurrent with a write-operation to the same register. Moreover *RdEnd* does now only restrict the returned value if there is no such $\perp$ value at the end of the read. This implements a Safe Register, since by definition, a Safe Register behaves as a Regular Register for reads that are not concurrent to a write and return some arbitrary value otherwise.

Reusing our model of a Regular Register for an Atomic Register is rather difficult. Since Atomic Registers are not used in the case study in Chapter 4 we restrict ourselves to a rough idea for a model of an Atomic Register. An atomic register stores an old value *old* and a new value *new*. A new value v' is written to a register Reg_{p_j} by setting *new* $:= v'$. Somewhere between the beginning and the end of a write the value in *new* overwrites the value in *old* (a new subaction is needed for that). A process that reads from a register Reg always applies the value from *old*. This will guarantee the properties of an Atomic Register.

[16]Isabelle/HOL theory: `Register.thy`, theorem(s): `ReadNewValuePossible,ReadOldValuePossible`

Using Regular Registers with DnAs

To use our subactions of for Regular Registers with DnAs we have to split the subactions into two parts: one that can be used within the enabled predicate of a DnA and one to be used in the transition function of the DnA. Since all subactions are deterministic, splitting the subaction is simple. We define the following atoms to be used in an enabled predicate of a DnA:

$$\begin{aligned}
\mathit{WriteBegin}_{en} &: \mathfrak{R} \times \mathbb{P} \to \mathit{bool} \\
\mathit{WriteEnd}_{en} &: \mathfrak{R} \times \mathbb{P} \to \mathit{bool} \\
\mathit{RdBegin}_{en} &: \mathfrak{R} \times \mathbb{P} \to \mathit{bool} \\
\mathit{RdEnd}_{en} &: \mathfrak{R} \times \mathbb{P} \times \mathbb{P} \times \mathcal{V}_{\text{reg}} \to \mathit{bool}
\end{aligned}$$

$$\begin{aligned}
\mathit{WriteBegin}_{en}(\mathbf{Sh}, p_i) &\triangleq \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rIdle} \\
&\quad \wedge \exists v \in \mathcal{V}_{\text{reg}} \,.\, \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i) = v \\
\mathit{WriteEnd}_{en}(\mathbf{Sh}, p_i) &\triangleq \exists v, v' \in \mathcal{V}_{\text{reg}} \,.\, \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i) = (v, v') \\
&\quad \wedge \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rWriting} \\
\mathit{RdBegin}_{en}(\mathbf{Sh}, p_i) &\triangleq \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rIdle} \\
\mathit{RdEnd}_{en}(\mathbf{Sh}, p_i, p_j, v) &\triangleq \mathit{rPhs} \triangleright \mathbf{Sh}(p_i) = \mathbf{rReading}_{p_j} \\
&\quad \wedge v \in \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)(p_j)
\end{aligned}$$

Each of these atoms is true if the correspondent subaction is enabled. The following atoms can then be used in the transition part of a DnA (as always we denote the second entry of a pair p by $2^{\text{nd}}(p)$):

$$\begin{aligned}
\mathit{WriteBegin}_{\Delta} &: \mathfrak{R} \times \mathbb{P} \times \mathcal{V}_{\text{reg}} \to \mathfrak{R} \\
\mathit{WriteEnd}_{\Delta} &: \mathfrak{R} \times \mathbb{P} \to \mathfrak{R} \\
\mathit{RdBegin}_{\Delta} &: \mathfrak{R} \times \mathbb{P} \times \mathbb{P} \to \mathfrak{R} \\
\mathit{RdEnd}_{\Delta} &: \mathfrak{R} \times \mathbb{P} \to \mathfrak{R}
\end{aligned}$$

$$\begin{aligned}
\mathit{WriteBegin}_{\Delta}(\mathbf{Sh}, p_i, v') &\triangleq \text{rwbupd}_{\mathbf{Sh}, p_i}^{(\text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i), v')} \\
\mathit{WriteEnd}_{\Delta}(\mathbf{Sh}, p_i) &\triangleq \mathbf{Sh}[p_i := (\mathbf{rIdle}, 2^{\text{nd}}(\text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i)), \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i))] \\
\mathit{RdBegin}_{\Delta}(\mathbf{Sh}, p_i, p_j) &\triangleq \mathbf{Sh}[p_i := (\mathbf{rReading}_{p_j}, \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i), \\
&\qquad\qquad \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)[p_j := \text{vset}\,(\text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_j))])] \\
\mathit{RdEnd}_{\Delta}(\mathbf{Sh}, p_i) &\triangleq \mathbf{Sh}[p_i := (\ \mathbf{rIdle},\ \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i),\ \text{view}^{\text{r}} \triangleright \mathbf{Sh}(p_i)\ )]
\end{aligned}$$

Finally we show that all subactions in σ_{sh} can be constructed from these atoms:

Theorem 2.3.7.5 (Subaction Reconstruction)
For all configurations $\mathfrak{C}, \mathfrak{C}' \in \mathbb{C}$, $p_i, p_j \in \mathbb{P}$ and all $v \in \mathcal{V}_{\text{reg}}$ the following assertions hold:

1. $\mathit{WriteBegin}(\boldsymbol{Sh}, \boldsymbol{Sh}', p_i, v) \equiv (\mathit{WriteBegin}_{en}(\boldsymbol{Sh}, p_i) \wedge \boldsymbol{Sh}' = \mathit{WriteBegin}_{\Delta}(\boldsymbol{Sh}, p_i, v))$
2. $\mathit{WriteEnd}(\boldsymbol{Sh}, \boldsymbol{Sh}', p_i) \equiv (\mathit{WriteEnd}_{en}(\boldsymbol{Sh}, p_i) \wedge \boldsymbol{Sh}' = \mathit{WriteEnd}_{\Delta}(\boldsymbol{Sh}, p_i))$
3. $\mathit{RdBegin}(\boldsymbol{Sh}, \boldsymbol{Sh}', p_i, p_j) \equiv (\mathit{RdBegin}_{en}(\boldsymbol{Sh}, p_i) \wedge \boldsymbol{Sh}' = \mathit{RdBegin}_{\Delta}(\boldsymbol{Sh}, p_i, p_j))$
4. $\mathit{RdEnd}(\boldsymbol{Sh}, \boldsymbol{Sh}', p_i, p_j, v) \equiv (\mathit{RdEnd}_{en}(\boldsymbol{Sh}, p_i, p_j, v) \wedge \boldsymbol{Sh}' = \mathit{RdEnd}_{\Delta}(\boldsymbol{Sh}, p_i))$

Proof. $\hookrightarrow$ Isabelle[17] and $\hookrightarrow$ Isabelle[18]

For a better understanding of the atoms for DnAs consider the following example. Assume we have a Distributed Algorithm that uses a module **Sh** for Regular Registers as described before. Furthermore in a state of a process only a natural number n is stored, therefore we have an array of states $\mathbf{S} : \mathbb{P} \to \mathbb{N}$. The algorithm uses the following process-action *Wr100*. If p_i's current value is less than 100, *Wr100* writes the value to the shared register of p_i and increments the value by 1:

$$\begin{aligned} \mathit{Wr100}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq (&\mathbf{S}_{\mathfrak{C}}(p_i) < 100 \\ &\wedge \mathit{WriteBegin}(\mathbf{Sh}_{\mathfrak{C}}, \mathbf{Sh}_{\mathfrak{C}'}, p_i, \mathbf{S}_{\mathfrak{C}}(p_i)) \\ &\wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{\mathfrak{C}}(p_i) + 1]) \end{aligned}$$

For working with DnAs we define functions C2LV^{100} and LV2C^{100} as follows:

$$\text{C2LV}^{100}(\mathfrak{C}, p_i) \triangleq (\mathbf{S}_{\mathfrak{C}}(p_i), \mathbf{Sh}_{\mathfrak{C}}, p_i)$$
$$\text{LV2C}^{100}(lv', p_i, \mathfrak{C}) \triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{lv'}] \\ \mathbf{Sh}_{lv'} \end{pmatrix}$$

Again, we denote the first entry of a Localview lv by $\mathbf{S}_{lv}$, and the second by $\mathbf{Sh}_{lv}$. The third entry is denoted by $\mathbf{ID}_{lv}$. Hence in its Localview a process p_i has access to its state $\mathbf{S}_{\mathfrak{C}}(p_i)$, the registers in $\mathbf{Sh}_{\mathfrak{C}}$ and to its identity. Note that since the complete module $\mathbf{Sh}_{\mathfrak{C}}$ is mapped to the Localview of p_i by C2LV^{100} and back into the configuration by LV2C^{100} we allow a process to read and write the whole content of the shared memory. This is due to the considerations in Section 2.3.4. Anyway manipulations of registers and other intended access to the module will be ruled out by the use of the provided atoms. Now we can define a corresponding DnA $\mathit{Wr100}^{\downarrow}$ for *Wr100*. $\mathit{Wr100}^{\downarrow}$ is the tuple $\mathit{Wr100}^{\downarrow} = (\ \mathit{Wr100}_{en},\ \mathit{Wr100}_{\Delta}\ )$ with:

$$\mathit{Wr100}_{en}(lv) \triangleq (\mathbf{S}_{lv} < 100 \wedge \mathit{WriteBegin}_{en}(\mathbf{Sh}_{lv}, \mathbf{ID}_{lv}))$$
$$\mathit{Wr100}_{\Delta}(lv) \triangleq (\mathbf{S}_{lv} + 1, \mathit{WriteBegin}_{\Delta}(\mathbf{Sh}_{lv}, \mathbf{ID}_{lv}, \mathbf{S}_{lv}), \mathbf{ID}_{lv})$$

$\mathit{Wr100}_{en}$ ensures that for the execution of $\mathit{Wr100}^{\downarrow}$ the *WriteBegin* subaction is enabled and the value in the state of p_i is less than 100. $\mathit{Wr100}_{\Delta}$ reflects the changes in the Localview of

[17]Isabelle/HOL theory: `Register.thy`, theorem(s): `RWBEqEnExecRWB,RWEEqEnExecRWE`
[18]Isabelle/HOL theory: `Register.thy`, theorem(s): `RRBEqEnExecRRB,RREEqEnExecRRE`

p_i caused by $\mathit{Wr100}^{\downarrow}$: the value in p_i's state is incremented by one and the register module is changed by the atom $\mathit{WriteBegin}_{\Delta}$. The identity of p_i is unchanged. Note that also in this case the definition of the DnA is much more compact compared to the definition of the lifted action.

Note that the atoms for *RdEnd* are given for theoretical considerations but have no practical relevance. This is due to the fact, that there can be no equivalent DnA for a non-deterministic action and, as described before, a proper use of *RdEnd* may not restrict the parameter v for the read value. On the one hand in a system with concurrent writes it is obviously intended that there is more than one option for v in term $\exists v \in \mathcal{V}_{\text{reg}} \,.\, \mathit{RdEnd}(\mathbf{Sh}, \mathbf{Sh}', p_i, p_j, v)$. On the other hand a deterministic use of *RdEnd* has to restrict the value v to exactly one value, but this contradicts the intention of the modelled read operation, because a process can not predict the value it is going to read. This yields a non-determinism if value v is used for the computation of the next configuration. Nevertheless regarding the atoms for *RdEnd*, we can construct a DnA which uses the atoms. Consider the following example that defines a DnA $\mathit{rdx}^{\downarrow}$. A process that executes $\mathit{rdx}^{\downarrow}$ reads a register of process p_x:

$$\begin{aligned}
\mathit{rdx}^{\downarrow} &\triangleq (\ \mathit{rdx}_{en},\ \mathit{rdx}_{\Delta}\ )\\
\mathit{rdx}_{en}(lv) &\triangleq \exists v \in \mathcal{V}_{\text{reg}} \,.\, \mathit{RdEnd}_{en}(\mathbf{Sh}_{lv}, \mathbf{ID}_{lv}, p_x, v)\\
\mathit{rdx}_{\Delta}(lv) &\triangleq (\mathbf{S}_{lv}, \mathit{RdEnd}_{\Delta}(\mathbf{Sh}_{lv}, \mathbf{ID}_{lv}), \mathbf{ID}_{lv})
\end{aligned}$$

This DnA is obviously deterministic and uses the atoms for RdEnd_{en} and RdEnd_{Δ}. But the variable v is locally bound in the definition of RdEnd_{en}. Therefore it cannot be used for calculations in RdEnd_{Δ}. A read action where the value read can not be utilised for calculations is of no practical relevance.

2.3.8. Failure Models

A crucial characteristic for a distributed algorithm is its ability to guarantee the desired properties in the presence of failures. A failure can be caused e.g. by a hardware defect or malicious intrusion and may result in malfunctions of the involved processes or can corrupt, reorder, duplicate or destroy messages that are not yet delivered. A common verification objective is to show that a certain algorithm is able to solve a problem even if a certain amount of failures occur. To do this kind of verification the possible failures must be covered by the model. Hence the model must comprise the expected faultless behaviour as well as a model for the possible cases of failure.

The problem of modelling failures is obvious: a failure is an undesirable behaviour of the system. Hence, if a failure occurs, it is not intended by the developer of the algorithm and therefore the consequences can not be anticipated in all details. Every failure model can therefore only capture a restricted view of the 'real' behaviour and for every fault-tolerant algorithm the failure class it tolerates has to be defined.

Well-known examples for failure models are ([Gä99], [GR06]):

- Crash-Failure (where processors simply stop their execution)

- Fail-Stop (where processors also crash but their neighbours can detect this easily)
- Fail-Silent (where process crashes are not reliably detected)
- Fail-Noisy (where crashes can be detected but not always in an accurate manner)
- Fail-Recovery (where crashed processes can recover)
- Byzantine (where faulty processors can behave arbitrarily)

[GR06] introduces these notions as classes for algorithms (not as failure models) and notes that these classes are not necessarily disjoint. Hence it is possible that certain algorithms are implemented to work in more than one of these models.

While an instance of the crash failure model will be used in Chapter 5.2. In Chapter 5.3 this model will be extended to an instance of the Fail-Recovery model that allows a resurrection of crashed processes.

Gärtner also introduces technical means to model failures. For a simple Crash Failure Model he proposes to apply the concept of [Aro92] to add virtual variables extending the actual state space of processes. These variables can then be used to indicate whether a process is up or crashed. The Crash Failure model can then be implemented by adding the condition that the process is not crashed as an additional guard to every action. This model for a crash is used for the algorithms in Section 5.1 and Section 5.3.

A variant of this model is used in Section 5.2. There we use an additional module **Crashed** (for modules cf. 2.3.1), which is simply a set of processes. A crash of a process is therefore modelled as the execution of the following process-action, which adds a non-crashed process to the set of crashed processes:

$$\mathit{Crash}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq p_i \notin \mathbf{Crashed}_{\mathfrak{C}} \wedge \mathbf{Crashed}_{\mathfrak{C}'} = \mathbf{Crashed}_{\mathfrak{C}} \cup \{ \ p_i \ \}$$

Of course, a process-action would normally not be enabled if the executing process is crashed. Therefore every definition of a process-action $\mathcal{A} \in \Phi$ should be of the kind:

$$\mathcal{A}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq p_i \notin \mathbf{Crashed}_{\mathfrak{C}} \wedge \dots$$

We can use a dual process-action *Recover* to model recovery of a process (i.e. an action that sets p_i alive again after it has crashed). Hence, such an action for recovery must only be enabled if p_i is crashed:

$$\mathit{Recover}(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq p_i \in \mathbf{Crashed}_{\mathfrak{C}} \wedge \mathbf{Crashed}_{\mathfrak{C}'} = \mathbf{Crashed}_{\mathfrak{C}} \setminus \{ \ p_i \ \}$$

The idea of recovery is exemplarily used for the algorithm in Section 5.3. Note that except for *Recover* and *Crash* all actions usually assert $\mathbf{Crashed}_{\mathfrak{C}'} = \mathbf{Crashed}_{\mathfrak{C}}$.

Note that the failure scenario might restrict the safety and liveness properties that can be guaranteed (definitions for certain safety and liveness properties will be given in Section 2.4).

Hence [Gä99] introduces the following terms to classify the different abilities of fault tolerant algorithms:

- Masking: An algorithm is called masking if it guarantees both safety and liveness for the respective failure model.
- Nonmasking: An algorithm is called nonmasking if it still guarantees liveness but not safety for the respective failure model.
- Fail safe: An algorithm is called nonmasking if it still guarantees safety but not liveness for the respective failure model.

Correct Processes

For the definition of requirements, it is often useful to divide correct and faulty processes with respect to a specific run. With $\text{Correct}(R)$ we denote the subset ($\text{Correct}(R) \subseteq \mathbb{P}$) of processes that are correct in run R. Since definitions of the term 'correct process' differ for different failure models we will give different definitions of correct processes for the different presented algorithms in the respective chapters.

2.4. Specifying Requirements

Requirements for algorithms can be defined as functional properties the considered algorithms must provide. Commonly the term properties refers to trace properties and hence in [AS86] properties are defined as sets of traces (in our terms a trace corresponds to a run). Note that this is indeed a sufficient definition for most applications in fault tolerant distributed computing. But some concerns, as for example security policies as *Noninterference*, can not be formulated by regarding only single runs (cf. [McL96]). Therefore extensions as e.g. *Hyperproperties* are introduced (cf. [CS10]). Since this work considers applications of fault tolerant computing we restrict ourselves to trace properties.

Alpern and Schneider ([AS86]) showed that every property can be decomposed into a safety property (informally: a property, which states that something bad does not happen [Lam77]) and a liveness property (informally: a property asserting that something good eventually happens [Lam77]). While for a violation of a safety property it is possible to give a finite prefix of a run as a witness, this is impossible for a liveness property. If the liveness property is satisfied it might suffice to give a finite prefix of a run as a witness. But for proving that a liveness property is violated we have to regard the entire possibly infinite run. For the verification of a program it is very important to distinguish between safety and liveness properties because the prove-strategies differ. While for safety properties arguments are based on invariants, for liveness properties well-foundedness arguments are needed ([AS86]). For a more comprehensive treatment and for references to the extensive literature on the subject of safety and liveness one may refer to [Kin94].

To formally define safety and liveness properties in our context, we need to introduce some operations on runs. Therefore let $\mathbb{T}^n \triangleq \{ m \in \mathbb{T} \mid m < n \}$ and $\mathbb{T}^\infty \triangleq \mathbb{T}$. For $n \in \mathbb{T}$ with Σ^n

we denote the set of all sequences of configurations

$$\Sigma^n \triangleq \{\ S : \mathbb{T}^n \rightarrow \mathbb{C}\ \}$$

with length n. With Σ^* we denote the set of all finite sequences of configurations:

$$\Sigma^* \triangleq \bigcup_{n \in \mathbb{T}} \Sigma^n.$$

And finally with Σ we denote the set:

$$\Sigma = \Sigma^* \cup \Sigma^\omega$$

We use $\text{len} : \Sigma \rightarrow \mathbb{T} \cup \{\ \infty\ \}$ as a function that returns the length of a sequence, hence for a sequence $s \in \Sigma$:

$$\text{len}(s) \triangleq \begin{cases} n & \text{, if } s \in \Sigma^n \text{ and } n \in \mathbb{T} \\ \infty & \text{, else} \end{cases}$$

Furthermore for a sequence $s \in \Sigma$ the domain $\text{dom}(s)$ is the time domain of s, i.e.

$$\text{dom}(s) = \begin{cases} \mathbb{T} & \text{, if } \text{len}(s) = \infty \\ \mathbb{T}^{\text{len}(s)} & \text{, else} \end{cases}$$

Now we define prefixes and extensions of sequences of configurations (adopted from [VVK05]):

Definition 2.4.0.1 (Prefixes, Extensions and Concatenation)
Let $r, r' \in \Sigma$ be two sequences of configurations.

1. *r is a **prefix** of r' (denoted by $r \sqsubseteq r'$) if and only if*

$$\text{dom}(r) \subseteq \text{dom}(r') \textit{ and } \forall m \in \text{dom}(r)\ .\ r(m) = r'(m).$$

2. *r is a **finite prefix** of r' if and only if*

$$r \sqsubseteq r' \textit{ and } \text{len}(r) \neq \infty.$$

3. *$[r \uparrow] \triangleq \{\ r' \in \Sigma \mid r \sqsubseteq r'\ \}$ is **the set of all extensions of** r.*

4. *For a finite sequence $r \in \Sigma^n$ ($n \in \mathbb{T}$) and a sequence $r' \in \Sigma^y$, we define **the concatenation** of r and r' as $r \cdot r' \in \Sigma^z$ with:*

$$z = \begin{cases} \infty, & \textit{if } y = \infty \\ n + y, & \textit{else} \end{cases}$$

$$r \cdot r'(m) = \begin{cases} r(m), & \textit{if } m < n \\ r'(m - n), & \textit{else} \end{cases}$$

Furthermore we use the notation $r_{\sqsubseteq n}$ to denote the finite prefix of r with length n.

A safety property P is a predicate where the following implication holds: if P is violated in a sequence s there is already a finite prefix $s_{\sqsubseteq n}$ such that for every extension of $s_{\sqsubseteq n}$ in $[s_{\sqsubseteq n}\uparrow]$ the property is violated. Formally (cf. [VVK05]):

Definition 2.4.0.2 (Safety property)
*A predicate $P : \Sigma \rightarrow$ **bool** is a safety property if and only if*

$$\forall s \in \Sigma \; . \; \neg P(s) \Rightarrow (\exists n \in \mathrm{dom}(s) \; . \; \forall s' \in [s_{\sqsubseteq n}\uparrow] \; . \; \neg P(s'))$$

We define a liveness property analogously to [AS86]:

Definition 2.4.0.3 (Liveness property)
*A predicate $P : \Sigma \rightarrow$ **bool** is a liveness property if and only if*

$$\forall s \in \Sigma^* \; . \; \exists s' \in \Sigma^\omega \; . \; P(s \cdot s')$$

Note that we define properties to be predicates on configuration sequences and not as sets of traces as [AS86]. Of course, these definitions are equivalent since if we consider a predicate $P : \Sigma \rightarrow \mathbf{bool}$ we can obtain the respective set of configuration sequences as $\{ \; s \in \Sigma \;\mid\; P(s) \; \}$. By using the predicate instead of the set, we have the characteristic function of the set at hand.

Lamport ([Lam94]) defines a state predicate to be a mapping from states to booleans. Since in our context states are configurations, we use the term *configuration predicate* for a mapping from configurations to booleans. We denote the set of all configuration predicates by CP:

$$CP \triangleq \mathbb{C} \rightarrow \mathbf{bool}$$

Now we can define a *configuration invariant*:

Definition 2.4.0.4 (Configuration invariant)
*A predicate $P : \Sigma \rightarrow$ **bool** is a Configuration invariant if and only if*

$$\exists Q \in CP \; . \; \forall s \in \Sigma \; . \; P(s) \leftrightarrow (\forall t \in \mathrm{dom}(s) \; . \; Q(s(t)))$$

Theorem 2.4.0.5
*Let $P : \Sigma \rightarrow$ **bool** be a Configuration invariant.*

1. *P is a safety property.*

2. *P is not a liveness property.*

Proof. $\hookrightarrow$ Isabelle[19]

Until now we only defined properties of sequences. To verify a Distributed Algorithm, we have to show that the desired properties hold for every execution (run) of the algorithm. Hence we define a property of an algorithm as follows:

[19]Isabelle/HOL theory: `ConfSeq.thy`, theorem(s): `ConfInvariantIsSafety, InvNoLiveness`

Definition 2.4.0.6 (Property)
Let $\mathfrak{A} = (\ \text{Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}}\ )$ *be an Algorithm.*

1. *A predicate* $P : \Sigma \rightarrow \textbf{bool}$ *is a property of* $\mathfrak{A}$ *if and only if*

$$\forall R \in \text{Runs}(\mathfrak{A})\ .\ P(R).$$

2. *A predicate* $P : \Sigma \rightarrow \textbf{bool}$ *is a property of* $\mathfrak{A}$ *under assumptions* $\mathcal{F}_{\text{ad}}$ *if and only if*

$$\forall R \in \text{Runs}_{\text{ad}}(\mathfrak{A})\ .\ P(R).$$

Since (obviously) for an Algorithm $\mathfrak{A}$ the set of Admissible Runs ($\text{Runs}_{\text{ad}}(\mathfrak{A})$) is a subset of all runs ($\text{Runs}(\mathfrak{A})$), every property P of $\mathfrak{A}$ is a property under assumptions $\mathcal{F}_{\text{ad}}$ for all (reasonable) sets $\mathcal{F}_{\text{ad}}$.

2.4.1. Requirements of Distributed Consensus

The problem we focus on in this thesis is the problem of Distributed Consensus. A distributed algorithm $\mathfrak{A}$ that solves Consensus must exhibit the four properties Validity, Agreement, Termination, and Irrevocability$N$ processes that have to agree on exactly one value. Therefore, an algorithm that solves Consensus is supposed to exhibit the follwing properties ([Lyn96], [CBS07] (cf. Section 1.4.1). To show that $\mathfrak{A}$ actually respects these properties we have to rewrite these properties in terms of our model. Regarding the mentioned properties, obviously the main concern is a local decision procedure of the single processes. Let $\mathbb{I}$ be the set of possible input values for the processes. Due to the Agreement property, if a process decides a value $v \in \mathbb{I}$, it must be sure that no other process decides some other value v' with $v' \neq v$. We assume that the decision value of a process p_i is encoded in its state $\mathbf{S}(p_i)$. We define a function $\text{dc} : \mathbb{S} \rightarrow \mathbb{I} \cup \{ \perp \}$ that extracts the decision from a process state. Hence if $\text{dc}(\mathbf{S}_{\mathfrak{C}}(p_i)) \neq \perp$ process p_i has decided a value in configuration $\mathfrak{C}$. If $\text{dc}(\mathbf{S}_{\mathfrak{C}}(p_i)) = \perp$, p_i has not yet decided in $\mathfrak{C}$. To respect the Validity property the decided value must have been an input value for one of the processes. Therefore we define a mapping $\text{v}_{\text{inp}} : \mathbb{P} \rightarrow \mathbb{I}$ which assigns an input value to every process.

Now we are able to formally redefine the properties of Consensus. Recall that we defined properties to be predicates with type $\Sigma \rightarrow \textbf{bool}$.
Validity holds for a run s if and only if for all t of the time domain of s whenever a process p_i's decision value v_t in configuration $s(t)$ is defined (formally: $v_t \neq \perp$) there is a process $p_k \in \mathbb{P}$ such that $\text{v}_{\text{inp}}(p_k) = v_t$.

$$\text{Validity}(s) \triangleq \forall t \in \text{dom}(s)\ .\ \forall p_i \in \mathbb{P}.\text{dc}(\mathbf{S}_{s(t)}(p_i)) \neq \perp \Rightarrow \\ \big(\exists p_k \in \mathbb{P}\ .\ \text{v}_{\text{inp}}(p_k) = \text{dc}(\mathbf{S}_{s(t)}(p_i))\big)$$

Agreement holds for a run s if and only if for all configurations $s(t)$ of s whenever for two processes $p_i, p_j \in \mathbb{P}$ decision values $v_{t,i}, v_{t,j}$ are defined then $v_{t,i} = v_{t,j}$.

$$\begin{aligned}\text{Agreement}(s) \triangleq \forall t \in \text{dom}(s) \ . \ \forall p_i, p_j \in \mathbb{P}.(\text{dc}(\mathbf{S}_{s(t)}(p_i)) \neq \bot \wedge \text{dc}(\mathbf{S}_{s(t)}(p_j)) \neq \bot \Rightarrow \\ \text{dc}(\mathbf{S}_{s(t)}(p_i)) = \text{dc}(\mathbf{S}_{s(t)}(p_j)))\end{aligned}$$

Termination holds for a run s if and only if for all processes $p_i \in \text{Correct}(s)$ the decision value of p_i is eventually defined.

$$\text{Termination}(s) \triangleq \forall p_i \in \text{Correct}(s) \ . \ \exists t \in \text{dom}(s) \ . \ \text{dc}(\mathbf{S}_{s(t)}(p_i)) \neq \bot$$

Irrevocability holds for a run s if and only if whenever for some configuration $s(t)$ $(t \in \text{dom}(s))$ of s the decision value v_t of a process $p_i \in \mathbb{P}$ is defined, then for all subsequent configurations $s(t')$ $(t' \in \text{dom}(s), t' \geq t)$ the decision value $v_{t'}$ will not be altered $(v_{t'} = v_t)$.

$$\begin{aligned}\text{Irrevocability}(s) \triangleq \forall t \in \text{dom}(s) \ . \ \forall p_i \in \mathbb{P}. \ \text{dc}(\mathbf{S}_{s(t)}(p_i)) \neq \bot \Rightarrow \\ \forall t' \in \text{dom}(s) \ . \ \big(t' \geq t \Rightarrow \text{dc}(\mathbf{S}_{s(t')}(p_i)) = \text{dc}(\mathbf{S}_{s(t)}(p_i))\big)\end{aligned}$$

Considering our definitions of Configuration invariants we observe:

Lemma 2.4.1.1

1. Validity *is a Configuration invariant.*
2. Agreement *is a Configuration invariant.*

Proof.$\hookrightarrow$ Isabelle[20]
With Theorem 2.4.0.5 this implies that Validity and Agreement are pure safety properties:

Corollary 2.4.1.2

1. Validity *is a safety property and not a liveness property.*
2. Agreement *is a safety property and not a liveness property.*

Proof.$\hookrightarrow$ Isabelle[21]
Although often mentioned and used, this result of course depends highly on the definition of Validity and Agreement. [CBTB00] gives an example for the effect of a slightly different definition: if for Agreement only correct processes are required to decide equal values, every prefix of a run where two processes decide for different values can be extended by a run where one of the two processes crashes and hence Charron-Bost et al. claim that Agreement is a liveness property.

A Configuration invariant is an assertion about what has to be true for every single configuration. Irrevocability makes assertions about a configuration and its successors. Hence it is an

[20]Isabelle/HOL theory: `ConfSeq.thy`, Lemma(s): `ValidityInv, AgreementInv`
[21]Isabelle/HOL theory: `ConfSeq.thy`, Corollaries: `Validity_Is_Safety, Agreement_Is_Safety`

assertion about more than one configuration and therefore it is no Configuration invariant. Nevertheless Irrevocability is a safety property: once we have a sequence $s \in \Sigma$ where Irrevocability is violated, by the definition of Irrevocability there are $t, t' \in \mathrm{dom}(s), p_i \in \mathbb{P}$ with $t \leq t'$ such that $\bot \neq \mathrm{dc}(s(t), p_i)$ and $\mathrm{dc}(s(t), p_i) \neq \mathrm{dc}(s(t'), p_i)$. Obviously all extensions of $s_{\sqsubseteq(t'+1)}$ violate Irrevocability and therefore:

Theorem 2.4.1.3
Irrevocability is a safety property.

Proof. $\hookrightarrow$ Isabelle[22]
Let s_d be a sequence where all processes eventually decide at time t_d and hence $\mathrm{dc}(s_d(t_d), p) \neq \bot$ for all $p \in \mathbb{P}$. For every finite sequence s that is extended by s_d Termination holds (formally: $\mathrm{Termination}(s \cdot s_d)$). With the definition of liveness properties we obtain:

Theorem 2.4.1.4
Termination *is a liveness property.*

Proof. $\hookrightarrow$ Isabelle[23]
Of course sometimes it is possible to have safety properties which are also liveness properties, but we also showed:

Theorem 2.4.1.5

1. Irrevocability *is not a liveness property.*

2. Termination *is not a safety property.*

Proof. $\hookrightarrow$ Isabelle[24]

[22]Isabelle/HOL theory: `ConfSeq.thy`, theorem(s): `Irrevocability_Is_Safety`
[23]Isabelle/HOL theory: `ConfSeq.thy`, theorem(s): `Termination_Is_Liveness`
[24]Isabelle/HOL theory: `ConfSeq.thy`, theorem(s): `IrrevocabilityNoLiveness,TerminationNoSafety`

2.5. Summarising Concepts of the Model

Before we will explain the strategies for the verification of a distributed algorithm, this section gives a summary of the introduced concepts of our model.

To model a distributed algorithm, we have to be aware what are the data structures the algorithm deals with. The mind map in Figure 2.11 illustrates the relations between the used data structures. Central to our concept is the data structure for a configuration. Therefore, the first step to model an algorithm is to define the domain of configurations $\mathbb{C}$. This means also to define the domain of process states $\mathbb{S}$ and to define all domains for the modules. This includes the decision for the used mechanism for interprocess communication (IPC).

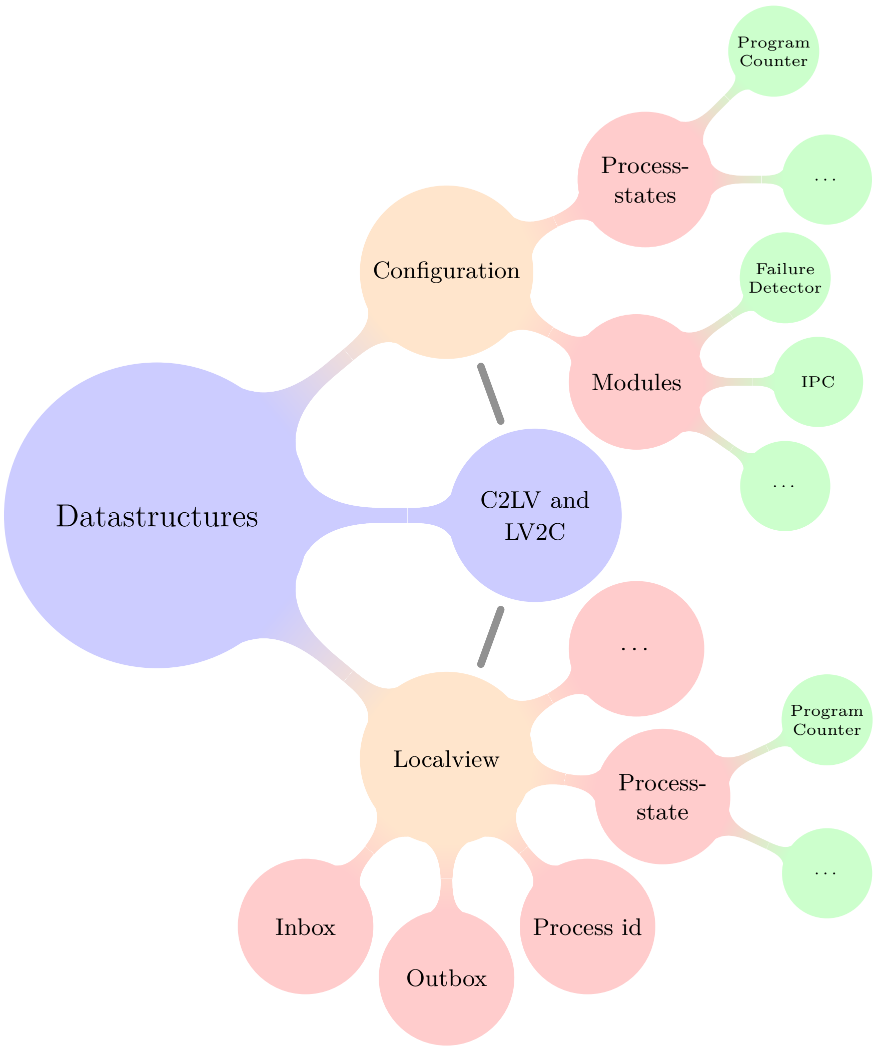

Figure 2.11.: Mindmap for Datastructures of the Model

For our concept of distribution, it is necessary to separate data a process is able to access from data the process cannot see. Therefore, we use the introduced concept of Localviews. A Localview $lv_{\mathfrak{C},p_i}$ for a process p_i and a configuration $\mathfrak{C}$ is a projection of $\mathfrak{C}$ to the part of $\mathfrak{C}$ a process p_i is able to access. This projection has to be given explicitly by the function C2LV : $\mathbb{C} \times \mathbb{P} \to \mathbb{L}$. To re-embed a Localview into a configuration, we define a dual function LV2C : $\mathbb{L} \times \mathbb{P} \times \mathbb{C} \to \mathbb{C}$, which takes a Localview lv, a process p_i and a configuration $\mathfrak{C}$ and returns a configuration $\mathfrak{C}'$ where lv has replaced the Localview of p_i in $\mathfrak{C}$. Usually, in a message passing environment, beside the process state, a Localview comprises the inbox and the outbox of a process. Of course this is different for other communication infrastructures. Additionally it may be required that processes know their (unique) process id.

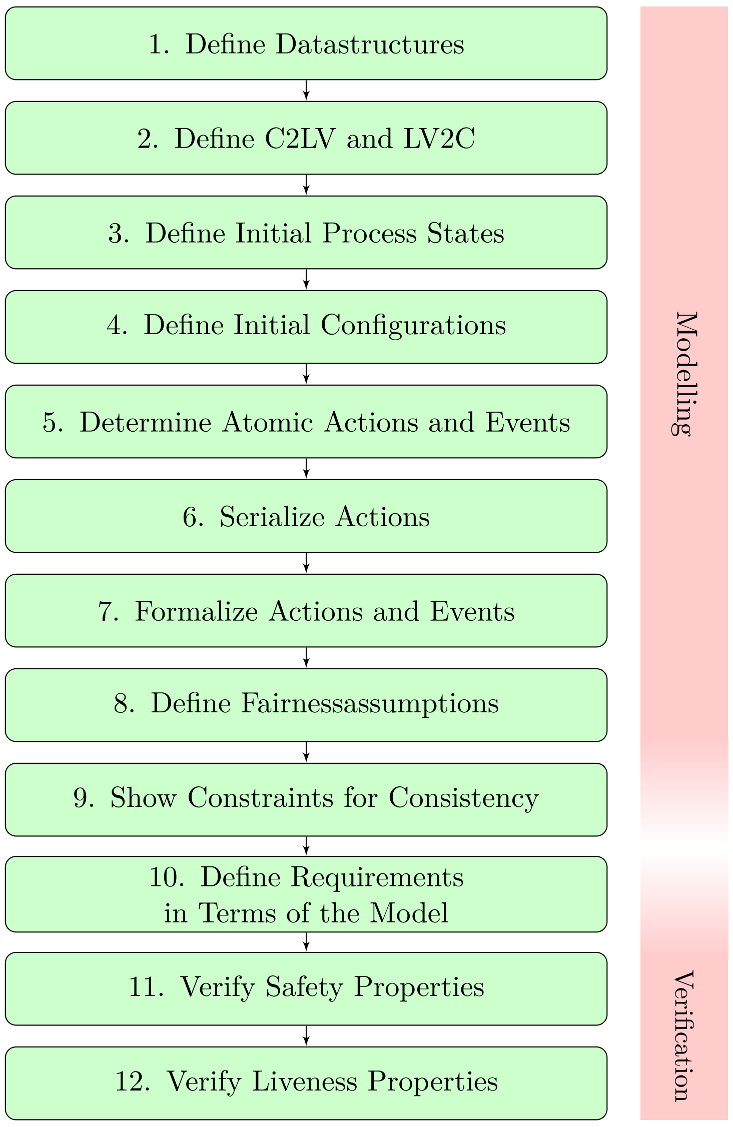
1. Define Datastructures
2. Define C2LV and LV2C
3. Define Initial Process States
4. Define Initial Configurations
5. Determine Atomic Actions and Events
6. Serialize Actions
7. Formalize Actions and Events
8. Define Fairnessassumptions
9. Show Constraints for Consistency
10. Define Requirements in Terms of the Model
11. Verify Safety Properties
12. Verify Liveness Properties
Modelling
Verification

Figure 2.12.: Process of Modelling and Verification

Figure 2.12 shows our modelling and verification process. After datastructures and functions C2LV and LV2C have been defined (Step 1 and 2), we have to fix the initial configurations (Step 4). Before we are able do this, it is necessary to define the initial process states (Step 3) for the single processes. When this is done, we can turn towards modelling the steps in our system. Firstly, we have to identify the atomic actions and events (Step 5) in our system. Therefore, our focus must be put on the communication protocol. While it may be possible to summarise local computation steps as a single atomic step, a main principle in distributed computing for inspecting concurrency problems is that different communication steps must be modelled as different atomic steps. That is necessary because the order of communication steps may influence the processes computations and therefore determines the sequence of configurations. Moreover the example of shared memory (see Section 2.3.7) shows that sometimes it is necessary to model a single read as multiple steps. Further examples and explanations for this will be given in our case study in Section 5.4. Usually the local behaviour of a process is given as a sequential algorithm. Therefore, process-actions have to respect the order in which actions are executed. Since our modelling approach uses a disjunction for selecting the next action to execute, we have to serialise process-actions by introducing phases and program counters which are used to define the conditions when an action is enabled (Step 6) (cf. [Fuz08]). Examples will be given in our case studies (see Chapter 5). In the next step we have to formally define the events and actions (Step 7).

Whereas events have always to be defined globally, with the introduction of non-lifted actions we introduced a concept to model transitions locally in a global context. Figure 2.13 depicts our advice how to decide for a formalisation of a process-action $\mathcal{A}$: If the action can be modelled deterministically, then the best way is to use our concept of DnAs and define a DnA $\mathcal{A}^{\downarrow}$ (cf. Section 2.3.3). For the entry Φ in $\mathfrak{A}$ we use DLift($\mathcal{A}^{\downarrow}$) such that DLift($\mathcal{A}^{\downarrow}$) $\in \Phi$. Due to the constraint LocalviewRespect, we have to give a non-lifted version $\mathcal{A}^{\downarrow}$ of $\mathcal{A}$ even if the action cannot be modelled deterministically. On the other hand, proofs are much simpler if we use the globally defined version of $\mathcal{A}$. Hence the recommended way here is to define both, $\mathcal{A}$ and the non-lifted version $\mathcal{A}^{\downarrow}$ and to show that $\mathcal{A} = \text{Lift}(\mathcal{A}^{\downarrow})$. In this case, we use $\mathcal{A}$ in Φ, such that $\mathcal{A} \in \Phi$. The non-lifted version $\mathcal{A}^{\downarrow}$ only has to be defined, to satisfy the constraint LocalviewRespect, which serves to guarantee that no process uses inaccessible global information. To make the entries in $\mathfrak{A}$ complete we have to define the fairness assumptions (Step 8) for our algorithm. In our experience these assumptions will often be revised later, since many subtle assumptions may not be recognised until they are needed for the proofs. The last modelling step is to show that the defined tuple $\mathfrak{A}$ satisfies the constraints locViewProp1, locViewProp2, LocalviewRespect, NoRemoteAccess and StateInvEv (Step 9).

The verification process starts with the definition of the requirements in terms of the model (Step 10, as explained in Section 2.4). Finally, the proofs have to be done. Our examples exhibit that it is often an advantage to start with the safety properties (Step 11). Firstly, many simple flaws in the model may be recognised instantly, while proving simple safety properties. Secondly, many simple invariants are of use when proving elaborated liveness properties (Step 12). The next chapter will explain the process of doing the proofs and introduce general proof strategies for distributed algorithms that are modelled within our framework.

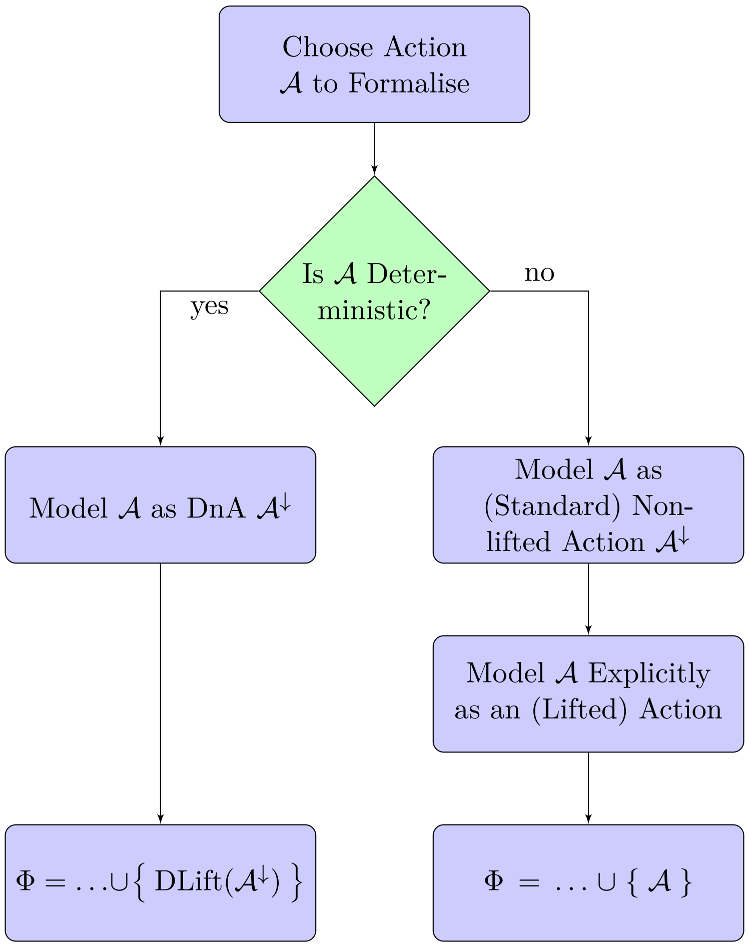
Choose Action $\mathcal{A}$ to Formalise
Is $\mathcal{A}$ Deterministic?
yes
no
Model $\mathcal{A}$ as DnA $\mathcal{A}^{\downarrow}$
Model $\mathcal{A}$ as (Standard) Non-lifted Action $\mathcal{A}^{\downarrow}$
Model $\mathcal{A}$ Explicitly as an (Lifted) Action
$\Phi = \ldots \cup \left\{ \text{DLift}(\mathcal{A}^{\downarrow}) \right\}$
$\Phi = \ldots \cup \{ \mathcal{A} \}$

Figure 2.13.: Modelling process-actions

3. Verifying Distributed Algorithms

This chapter outlines schemata for the most important proof-techniques that are used during the verification of different distributed algorithms (cf. case studies in Chapter 5). The respective applied technique depends on the kind of property that must be verified. As will be demonstrated, the strategies used to verify safety properties differ from applied techniques for verifying liveness properties. This is in compliance with a well known result of Alpern and Schneider [AS86], which states that verifying a safety property requires an invariance argument while for showing a liveness property a well-foundness argument is needed.

We will present proof-techniques at a general level and introduce examples for their application for the problem of Distributed Consensus.

3.1. 'Inspection of the Code'

As explained in [KNR12] to show that an algorithm exhibits certain properties, we have to refer to its 'code'. On the level of pseudo code this kind of reference cannot be formal because pseudo code (by definition) has no formal semantics. The reader is to believe that certain basic assertions are implied by the presented lines of code; e.g., if line 27 states $x := 5$ then, after line 27 is executed by p_i, variable x will indeed have value 5. Reasoning is done by 'inspection of the code'. Such scenarios seem very simple and the reasoning convincing as long as we deal with sequential programs. For distributed algorithms, this kind of local reasoning is, of course, error-prone, because one might assume $x = 5$ when executing line 28, which might be wrong if x is shared and another process changes x while p_i moves from line 27 to 28. In [FMN07], the reference to pseudo code is replaced by the reference to formally-defined global transition rules. Then, if some action $\mathcal{A}_x$ is provably the only one that changes the variable x of process p_i and process p_i's variable has changed from time t_x to t then, obviously, "by inspection of" the rules, we can infer that $\mathcal{A}_x$ was executed between t_x and t. This concept for reasoning is a useful basis for the later application within a theorem prover.

In the context of configurations, we use the term 'component' for a mapping $cp : \mathbb{C} \rightarrow \alpha$, where α denotes an arbitrary type (in Isabelle α is a type variable, i.e. a variable that can be instantiated with a concrete type when we are dealing with a concrete component). Hence in the following a component is an arbitrary function $cp : \mathbb{C} \rightarrow \alpha$, where α is an arbitrary type.

Definition 3.1.0.6 (Component preserving Actions and Events)
Let $\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}}\ )$ *be an algorithm and* $c : \mathbb{C} \rightarrow \alpha$ *be a component.*
An action $\mathcal{A} \in \Phi$ **preserves** *component c if and only if*

$$\forall p_i \in \mathbb{P}\ .\ \forall \mathfrak{C}, \mathfrak{C}' \in \mathbb{C}\ .\ \left(\mathcal{A}(\mathfrak{C}, \mathfrak{C}', p_i) \Rightarrow c(\mathfrak{C}) = c(\mathfrak{C}')\right)$$

An event $\mathcal{A} \in \Psi$ **preserves** *component* c *if and only if*

$$\forall \mathfrak{C}, \mathfrak{C}' \in \mathbb{C} \ . \ (\mathcal{A}(\mathfrak{C}, \mathfrak{C}') \Rightarrow c(\mathfrak{C}) = c(\mathfrak{C}'))$$

With $\Xi_\Phi(c) \subseteq \Phi$ *we denote the set of process* $-$ *actions that do not preserve component* c.
With $\Xi_\Psi(c) \subseteq \Psi$ *we denote the set of events that do not preserve component* c.
And finally with $\Xi(c)$ *we denote* $\Xi_\Phi(c) \cup \Xi_\Psi(c)$.

In our introductory example $\mathcal{A}_x$ does not preserve component x.
To show that certain subsets $A \subseteq (\Phi \cup \Psi)$ preserve a component c can help to reduce the effort for proving properties dealing with c.

For example, let us assume that $c : \mathbb{C} \rightarrow \mathbb{N}$ is a component of the configurations of an algorithm $\mathfrak{A} = (\text{ Init}, \ \Phi, \ \Psi, \ \mathcal{F}_{\text{ad}} \ )$ and $R \in \text{Runs}(\mathfrak{A})$ is a run of $\mathfrak{A}$. Furthermore, let us assume that we want to show that the implication $(c(R(t)) = 4) \Rightarrow (c(R(t+1)) = 4)$ holds for all $t \in \mathbb{T}$. Since R is a run we know that there is a step $R(t) \rightarrow R(t+1)$ or $\text{FinSt}(R(t), R(t+1))$. The final stuttering case is trivial because $R(t) = R(t+1)$ holds. So, let us consider the case where there is a step $R(t) \rightarrow R(t+1)$. $R(t) \rightarrow R(t+1)$ implies there is an $\mathcal{A}$ such that $\exists p_i \in \mathbb{P} \ . \ \mathcal{A} \in \Phi \wedge \mathcal{A}(R(t), R(t+1), p_i)$ or $\mathcal{A} \in \Psi \wedge \mathcal{A}(R(t), R(t+1)))$. Hence, $\mathcal{A} \in (\Phi \cup \Psi)$, and since $\mathfrak{A}$ is an algorithm we know that $(\Phi \cup \Psi)$ is finite. This yields a case distinction on $\mathcal{A}$ with $|\Phi| + |\Psi|$ many cases. But, if we know that there are only two actions that do not preserve c, we have to only consider these two actions since all other actions would imply $c(R(t)) = c(R(t+1))$. Therefore we can reduce the cases to the relevant set of actions $\Xi(c)$, as we would have done in a paper proof by telling the reader that for all other actions the implication $(c(R(t)) = 4) \Rightarrow (c(R(t+1)) = 4)$ holds obviously since they do not change $c(R(t)$. This is summarized by the following theorem:

Theorem 3.1.0.7
Let $\mathfrak{A} = (\text{ Init}, \ \Phi, \ \Psi, \ \mathcal{F}_{\text{ad}} \ )$ *be an algorithm, let* $c : \mathbb{C} \rightarrow \alpha$ *be a component of the configurations,* $P : \alpha \rightarrow$ ***bool*** *a predicate on* α, $R \in \text{Runs}(\mathfrak{A})$ *and* $t \in \mathbb{T}$.

If $\quad \forall \mathcal{A} \in \Xi_\Phi(c) \ . \ \forall p_i \in \mathbb{P} \ . \ (\mathcal{A}(R(t), R(t+1), p_i) \wedge P(c(R(t))) \Rightarrow P(c(R(t+1))))$
and $\quad \forall \mathcal{A} \in \Xi_\Psi(c) \ . \ (\mathcal{A}(R(t), R(t+1)) \wedge P(c(R(t))) \Rightarrow P(c(R(t+1))))$
then

$$P(c(R(t))) \Rightarrow P(c(R(t+1)))$$

Proof. $\hookrightarrow$ Isabelle[1]

Proposition 3.1.0.8
Let $\mathfrak{A} = (\text{ Init}, \ \Phi, \ \Psi, \ \mathcal{F}_{\text{ad}}, \ \text{C2LV}, \ \text{LV2C} \ )$ *be a Distributed Algorithm and let* $c_{\boldsymbol{S}} : \mathbb{C} \rightarrow (\mathbb{P} \rightarrow \mathbb{S})$ *be the function* $c_{\boldsymbol{S}}(\mathfrak{C}) \triangleq \boldsymbol{S}_{\mathfrak{C}}$ *that extracts the array of process states from a given configuration. Every event* $\mathcal{A} \in \Psi$ *preserves* $c_{\boldsymbol{S}}$.

Proof. $\hookrightarrow$ Isabelle[2]

[1]Isabelle/HOL theory: `DistributedAlgorithm.thy`, theorem(s): `NonCompPreservingActionsMatter`
[2]Isabelle/HOL theory: `DistributedAlgorithm.thy`, Lemma(s): `EventspreserveStates`

Hence, whenever we have to verify a property that is only an assertion about local process states, we do not need to consider events but only process-actions in the respective case distinction. With Theorem 3.1.0.7 we get:

Corollary 3.1.0.9
Let $\mathfrak{A} = (\text{ Init}, \ \Phi, \ \Psi, \ \mathcal{F}_{\text{ad}} \ )$ *be an algorithm,*
let $c_{\boldsymbol{S}} : \mathbb{C} \rightarrow (\mathbb{P} \rightarrow \mathbb{S})$ *be the function* $c_{\boldsymbol{S}}(\mathfrak{C}) \triangleq \boldsymbol{S}_{\mathfrak{C}}$ *that extracts the array of process states from a given configuration,* $P : (\mathbb{P} \rightarrow \mathbb{S}) \rightarrow$ ***bool*** *a predicate on process arrays,* $R \in \text{Runs}(\mathfrak{A})$ *and* $t \in \mathbb{T}$.

If $\quad \forall \mathcal{A} \in \Xi_{\Phi}(c_{\boldsymbol{S}}) \ . \ \forall p_i \in \mathbb{P} \ . \ \mathcal{A}(R(t), R(t+1), p_i) \wedge P(c_{\boldsymbol{S}}(R(t))) \Rightarrow P(c_{\boldsymbol{S}}(R(t+1)))$
then

$$P(c_{\boldsymbol{S}}(R(t))) \Rightarrow P(c_{\boldsymbol{S}}(R(t+1)))$$

Proof.$\hookrightarrow$ Isabelle[3]
Hence, if we want to verify a property $P(c_{\mathbf{S}}(R(t))) \Rightarrow P(c_{\mathbf{S}}(R(t+1)))$ by application of Corollary 3.1.0.9, we only have to consider the cases for the process-actions.

By Lemma 2.3.2.4 p_i's local state can only be changed by p_i in a distributed algorithm. Therefore, to show an assertion about the state of p_i, it suffices to consider the cases where p_i executes a $process - action$.

Corollary 3.1.0.10
Let $\mathfrak{A} = (\text{ Init}, \ \Phi, \ \Psi, \ \mathcal{F}_{\text{ad}} \ )$ *be a distributed algorithm,*
let $c_{\boldsymbol{S}} : \mathbb{C} \rightarrow (\mathbb{P} \rightarrow \mathbb{S})$ *be the function* $c_{\boldsymbol{S}}(\mathfrak{C}) \triangleq \boldsymbol{S}_{\mathfrak{C}}$ *that extracts the array of process states from a given configuration and ,* $P : \mathbb{S} \rightarrow$ ***bool*** *a predicate on* $\mathbb{S}$, $R \in \text{Runs}(\mathfrak{A})$ *and* $t \in \mathbb{T}$.

If $\quad \forall \mathcal{A} \in \Xi_{\Phi}(c_{\boldsymbol{S}}) \ . \ \mathcal{A}(R(t), R(t+1), p_i) \wedge P_x(c_{\boldsymbol{S}(p_i)}(R(t))) \Rightarrow P_x(c_{\boldsymbol{S}(p_i)}(R(t+1)))$
then

$$P(c_{\boldsymbol{S}(p_i)}(R(t))) \Rightarrow P(c_{\boldsymbol{S}(p_i)}(R(t+1)))$$

where for a configuration $\mathfrak{C}$ *and* $p_i \in \mathbb{P}$ *the term* $c_{\boldsymbol{S}(p_i)}(\mathfrak{C})$ *is defined as* $c_{\boldsymbol{S}(p_i)}(\mathfrak{C}) \triangleq (c_{\boldsymbol{S}}(\mathfrak{C}))(p_i)$ *(the entry of* p_i *in the array of states).*

Proof.$\hookrightarrow$ Isabelle[4]
The following section will explain why assertions of the kind $P(c(R(t))) \Rightarrow P(c(R(t+1)))$ are extremely relevant.

3.2. Invariant-Based Reasoning

In the previous section we showed how to deal with assertions of the form $P(c(R(t))) \Rightarrow P(c(R(t+1)))$. In this section we make use of these kind of assertions to prove invariants. Let us assume we want to verify an arbitrary Configuration invariant P_{inv}. By definition of a

[3]Isabelle/HOL theory: `DistributedAlgorithm.thy`, Corollary: `NonStPreservingPAMatter`
[4]Isabelle/HOL theory: `DistributedAlgorithm.thy`, Corollary: `NonStPreservingPAMatter2`

Configuration invariant (Definition 2.4.0.4) there is a $Q \in CP$ such that for a sequence $s \in \Sigma$ invariant $P_{\text{inv}}(s)$ can be represented by

$$\forall t \in \text{dom}(s) \,.\, Q(s(t)). \tag{PINV}$$

To show that P_{inv} is a property of a distributed algorithm $\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}}\ )$ we have to show

$$\forall R \in \text{Runs}(\mathfrak{A})\ \,.\, P_{\text{inv}}(R).$$

By definition of a run (runs are always infinite) the domain of a run is $\mathbb{T}$. Hence, with PINV we have to show that for an arbitrary run R the assertion

$$\forall t \in \mathbb{T} \,.\, Q(R(t))$$

holds. Of course, this can be easily done by induction using the following induction scheme:

$$Q(R(0)) \wedge (\forall t \in \mathbb{T} \,.\, Q(R(t)) \Rightarrow Q(R(t+1))) \quad \Rightarrow \quad \forall t \in \mathbb{T} \,.\, Q(R(t))$$

To show $\forall t \in \mathbb{T} \,.\, Q(R(t))$ we have to prove:

1. $Q(R(0))$
2. $\forall t \in \mathbb{T} \,.\, Q(R(t)) \Rightarrow Q(R(t+1))$

The first will usually be implied by the Init predicate, the latter is an assertion of the kind we discussed in Section 3.1. Hence, for this technique, the presented formal version of 'inspection of the code' is pertinent.

This strategy to find a proof that a Configuration invariant is a property of a distributed algorithm, leads to the standard proof technique of induction over time t, i.e., along the configurations of a run (cf. [KNR12]).

For example let us consider the sketch of a proof for Validity for a Distributed Consensus algorithm $\mathfrak{A} = (\text{Init}, \{\ \mathcal{A}_1,\ \ldots,\ \mathcal{A}_k\ \}, \{\ \mathcal{A}_l,\ \ldots,\ \mathcal{A}_m\ \}, \mathcal{F}_{\text{ad}})$. Recall the definition of Validity:

$$\text{Validity}(R) = \forall t \in \mathbb{T} \,.\, \forall p_i \in \mathbb{P}.(\text{dc}(R(t), p_i) \neq \bot \Rightarrow (\exists p_k \in \mathbb{P} \,.\ \text{v}_{\text{inp}}(p_k) = \text{dc}(R(t), p_i)))$$

We can define $Q_V \in CP$:

$$Q_V(\mathfrak{C}) = \forall p_i \in \mathbb{P}.(\text{dc}(\mathfrak{C}, p_i) \neq \bot \Rightarrow (\exists p_k \in \mathbb{P} \,.\ \text{v}_{\text{inp}}(p_k) = \text{dc}(\mathfrak{C}, p_i)))$$

such that we have

$$\text{Validity}(R) = \forall t \in \mathbb{T} \,.\, Q_V(R(t)).$$

Applying the induction scheme for Validity means to show:

1. $Q_V(R(0))$
2. $\forall t \in \mathbb{T} \,.\, Q_V(R(t)) \Rightarrow Q_V(R(t+1))$

A Consensus algorithm where processes start with a predetermined decision would be useless. Hence, we assume that the predicate Init implies $\forall p_i \in \mathbb{P} \;.\; \mathrm{dc}(\mathbf{S}_{R(0)}(p_i)) = \bot$. This implies $Q_V(R(0))$. Therefore, it remains to show $\forall t \in \mathbb{T} \;.\; Q_V(R(t)) \Rightarrow Q_V(R(t+1))$.
Assume

$$Q_V(R(t)) \tag{A1}$$

for an arbitrary time $t \in \mathbb{T}$. Using the definition of Q_V, we have to show $Q_V\,(R\,(t+1))$:

$$\forall p_i \in \mathbb{P}.\Big(\mathrm{dc}(\mathbf{S}_{R(t+1)}(p_i)) \neq \bot \Rightarrow \big(\exists p_k \in \mathbb{P} \;.\; \mathrm{v_{inp}}(p_k) = \mathrm{dc}(\mathbf{S}_{R(t+1)}(p_i))\big)\Big).$$

Assume $p_i \in \mathbb{P}$ is an arbitrary process. Let $P_V : \mathbb{S} \to \mathbf{bool}$ be the predicate

$$P_V(S_x) \triangleq (\mathrm{dc}(S_x) \neq \bot \Rightarrow (\exists p_k \in \mathbb{P} \;.\; \mathrm{v_{inp}}(p_k) = \mathrm{dc}(S_x))). \tag{PVDEF}$$

From A1 we have $\forall p_i \in \mathbb{P} \;.\; P_V(\mathbf{S}_{R(t)}(p_i))$ and hence $P_V(\mathbf{S}_{R(t)}(p_i))$ and it remains to show that $P_V(\mathbf{S}_{R(t+1)}(p_i))$. Putting it all together, we have to show the implication

$$P_V(\mathbf{S}_{R(t)}(p_i)) \Rightarrow P_V(\mathbf{S}_{R(t+1)}(p_i))$$

where P_V is a predicate on $\mathbb{S}$. By Corollary 3.1.0.10 we get the conjecture by showing

$$\forall \mathcal{A} \in \Xi_\Phi(c_\mathbf{S}) \;.\; \mathcal{A}(R(t), R(t+1), p_i) \wedge P_V(c_{\mathbf{S}(p_i)}(R(t))) \Rightarrow P_V(c_{\mathbf{S}(p_i)}(R(t+1)))$$

where $c_\mathbf{S} : \mathbb{C} \to (\mathbb{P} \to \mathbb{S})$ is the function $c_\mathbf{S}(\mathfrak{C}) \triangleq \mathbf{S}_\mathfrak{C}$ that extracts the array of process states from a given configuration. Hence, if we are lucky and there is only one action $\mathcal{A}_{\mathrm{decide}}$ that changes the decision value (i.e. $\Xi_\Phi(c_\mathbf{S}) = \{\ \mathcal{A}\ \}$), we only have to show that this action preserves the invariant, i.e.

$$\mathcal{A}_{\mathrm{decide}}(R(t), R(t+1), p_i) \wedge P_V(c_{\mathbf{S}(p_i)}(R(t))) \Rightarrow P_V(c_{\mathbf{S}(p_i)}(R(t+1)))$$

Otherwise (if $|\Xi_\Phi(c_\mathbf{S})| > 1$) we have to show $|\Xi_\Phi(c_\mathbf{S})|$ many cases.

We manifest this strategy by formulating a general induction theorem. It states that to show an assertion $P : \alpha \to \mathbf{bool}$ on a component $c : \mathbb{C} \to \alpha$, it suffices to show that every action and event that does not preserve component c preserves the value of P:

Theorem 3.2.0.11 (CPInduct)
Let $\mathfrak{A} = (\ \mathrm{Init},\ \Phi,\ \Psi\ )$ be a Safe Algorithm, R be a run of $\mathfrak{A}$ and $c : \mathbb{C} \to \alpha$ be a component of the configurations.

$$\forall t \in \mathbb{T} \;.\; P\,(c\,(R\,(t)))$$

is implied by the conjunction of the following assertions:

Base: $P(c(R(0)))$

I1: $\forall t \in \mathbb{T} \;.\; \forall p_i \in \mathbb{P} \;.\; \forall \mathcal{A} \in \Xi_\Phi \;.\; (R\,(t) \rightarrow_{p_i:\mathcal{A}} R\,(t+1)) \wedge P\,(c\,(R\,(t))) \Rightarrow P\,(c\,(R\,(t+1)))$

I2: $\forall t \in \mathbb{T} \;.\; \forall \mathcal{A} \in \Xi_\Psi \;.\; (R\,(t) \rightarrow_{\mathrm{ev}:\mathcal{A}} R\,(t+1)) \wedge P\,(c\,(R\,(t))) \Rightarrow P\,(c\,(R\,(t+1)))$

Proof. $\hookrightarrow$ Isabelle[5]

This technique is evidently a suitable strategy to prove Validity and Agreement and was used to prove many further configuration invariants. In the following, we present a method to handle safety properties that are no Configuration invariants. Therefore, let us examine the case for Irrevocability:

$$\begin{aligned}\text{Irrevocability}(s) \quad & \triangleq \forall t \in \text{dom}(s)\ .\ \forall p_i \in \mathbb{P}.\ \text{dc}(\mathbf{S}_{s(t)}(p_i)) \neq \bot \Rightarrow \\ & \big(\forall t' \in \text{dom}(s)\ .\ \big(t' \geq t \Rightarrow \text{dc}(\mathbf{S}_{s(t')}(p_i)) = \text{dc}(\mathbf{S}_{s(t)}(p_i))\big)\big)\end{aligned}$$

To prove that every run $R \in \text{Runs}(\mathfrak{A})$ satisfies Irrevocability we consider an arbitrary time $t \in \mathbb{T}$ and an arbitrary process $p_i \in \mathbb{P}$ and prove:

$$\begin{aligned}\text{Irrevocability}_{t,p_i}(R) \quad & \triangleq \text{dc}(\mathbf{S}_{R(t)}(p_i)) \neq \bot \Rightarrow \\ & \big(\forall t' \in \mathbb{T}\ .\ \big(t' \geq t \Rightarrow \text{dc}(\mathbf{S}_{R(t)}(p_i)) = \text{dc}(\mathbf{S}_{R(t')}(p_i))\big)\big)\end{aligned}$$

At first, let us inspect the general characteristics of formula $\text{Irrevocability}_{t,p_i}$. The premise of $\text{Irrevocability}_{t,p_i}$ is a condition b on a component $c : \mathbb{C} \to \alpha$. After $b(c(R(t)))$ is fulfilled once in configuration $R(t)$, some assertion $c(R(t)) = c(R(t'))$ has to be true for all successive configurations $R(t')$. Hence, we are looking for a proof of a formula with the following pattern:

$$b(c(R(t))) \Rightarrow (\forall t' \in \mathbb{T}\ .\ t' \geq t \Rightarrow c(R(t)) = c(R(t'))) \tag{Goal}$$

Since this is true if and only if $c(R(t)) = c(R(t+1)) = c(R(t+2)) = \ldots$ it suffices to show the proposition

$$b(c(R(t'))) \Rightarrow c(R(t')) = c(R(t'+1)). \tag{Step}$$

for an arbitrary time $t' \in \mathbb{T}$ and an arbitrary run $R \in \text{Runs}(\mathfrak{A})$. Of course Step can be proved again by case distinction on the action (respectively event) that happens from t' to $t'+1$ and the same techniques as for Configuration invariants for reducing the cases can be applied. After proving Step we can prove Goal, i.e. we show that the following theorem holds:

Theorem 3.2.0.12

*Let $s \in \Sigma^\omega$, $c : \mathbb{C} \to \alpha$ be a component of configurations, let $b : \alpha \to$ **bool** be a predicate and $t \in \mathbb{T}$.*

$$\forall t' \in \mathbb{T}\ .\ b(c(s(t'))) \Rightarrow c(s(t')) = c(s(t'+1)) \tag{Step}$$

implies

$$b(c(s(t))) \Rightarrow (\forall t' \in \mathbb{T}\ .\ t' \geq t \Rightarrow c(s(t)) = c(s(t'))) \tag{Goal}$$

Proof. We show $t' \in \mathbb{T} \wedge t' \geq t \Rightarrow c(s(t)) = c(s(t'))$ under the assumption

$$b(c(s(t))) \tag{A1}$$

using Step by induction on t'. Let us assume we have $t' \in \mathbb{T}$ and $t' \geq t$.

[5]Isabelle/HOL theory: `DistributedAlgorithm.thy`, theorem(s): `CPInduct`

Base case $t' = 0$: With $t' = 0$ we have $0 = t' = t$. Hence, $c(s(t)) = c(s(0) = c(s(t')$.

Inductive step: For induction hypothesis assume

$$t' \in \mathbb{T} \wedge t' \geq t \Rightarrow c(s(t)) = c(s(t')). \tag{IH}$$

We have to show

$$((t'+1)) \in \mathbb{T} \wedge ((t'+1) \geq t) \Rightarrow c(s(t)) = c(s(t'+1)).$$

Hence, assume $t'+1 \geq t$. For the case where $t'+1 = t$ the proposition $c(s(t)) = c(s(t'+1))$ is trivial. Hence, consider the case where $t'+1 > t$. Since $t', t \in \mathbb{T}$ this implies $t' \geq t$ and with IH we get $c(s(t)) = c(s(t'))$. With A1 we get $b(c(s(t')))$. Furthermore with Step we have $c(s(t')) = c(s(t'+1))$. Since $c(s(t)) = c(s(t'))$ we finally have $c(s(t)) = c(s(t'+1))$.

$\Box$(see also $\hookrightarrow$ Isabelle[6])
For proving Irrevocability$_{t,p_i}(R)$, we define $c_V : \mathbb{C} \rightarrow (\mathbb{I} \cup \{\perp\})$ with $c_V(\mathfrak{C}) \triangleq \mathrm{dc}(\mathbf{S}_{\mathfrak{C}}(p_i))$ and $b_V : \mathbb{I} \rightarrow \mathbf{bool}$ as $b_V(x) \triangleq x \neq \perp$.
By application of Theorem 3.2.0.12 it suffices to show

$$\forall t' \in \mathbb{T} \,.\, b_V(c_V(R(t'))) \Rightarrow c_V(R(t')) = c_V(R(t'+1)) \tag{IrreStep}$$

for proving

$$b_V(c_V(R(t))) \Rightarrow (\forall t' \in \mathbb{T} \,.\, t' \geq t \Rightarrow c_V(R(t)) = c_V(R(t')))\,. \tag{IrreGoal}$$

Finally, by the defintions of b_V and c_V we have

$$\begin{aligned}
& b_V(c_V(R(t))) \Rightarrow (\forall t' \in \mathbb{T} \,.\, t' \geq t \Rightarrow c_V(R(t)) = c_V(R(t'))) \\
\overset{\text{Def. } b_V}{\equiv}\; & c_V(R(t)) \neq \perp \Rightarrow (\forall t' \in \mathbb{T} \,.\, t' \geq t \Rightarrow c_V(R(t)) = c_V(R(t'))) \\
\overset{\text{Def. } c_V}{\equiv}\; & \mathrm{dc}(\mathbf{S}_{R(t)}(p_i)) \neq \perp \Rightarrow \\
& \quad \big(\forall t' \in \mathbb{T} \,.\, t' \geq t \Rightarrow \mathrm{dc}(\mathbf{S}_{R(t)}(p_i)) = \mathrm{dc}(\mathbf{S}_{R(t')}(p_i))\big) \\
\overset{\text{Def. Irrevocability}_{t,p_i}}{\equiv}\; & \mathrm{Irrevocability}_{t,p_i}(R)
\end{aligned}$$

Hence, to prove Irrevocability it suffices to prove IrreStep which can be done by case distinction on actions and events from t' to $t'+1$ and the techniques presented to reduce the case distinctions (see Section 3.1).

[6]Isabelle/HOL theory: `DistributedAlgorithm.thy`, theorem(s): `CondGe`

3.3. History-Based Reasoning

[KNR12] adressed the problem that reasoning along the timeline gets more difficult if assertions about the past are included. Showing that p_i received a message m on its way to configuration $\mathfrak{C}$ would require inspecting every possible prefix of a run unless there is some kind of bookkeeping implemented in the model. In [FMN07], this problem is solved by the introduction of history variables [Cli73, Cla78, Cli81] that keep track of events during the execution of the algorithm. For verification purposes of concurrent programs, history variables are common (see [GL00, Owi76]). Technically, we make history variables an explicit part of our model that also serves for the needed Interprocess Communication (IPC). This provides access to the entire communication history. Every sent message is stored in the history and will not be deleted during a run. Hence, when inspecting a configuration $R(t)$, all messages sent before t are accessible. Therefore, the above-mentioned assertion can be reduced to the simple check that m is in the message history of $\mathfrak{C}$.

Let us again consider the example of Validity. To prove Validity, we have to show that every value decided by a process p_i has been an input value of a process p_k. Hence, a standard proof would start with an assumption that an arbitrary process p_i decided a value v_i and would then constrain that v_i must have been an input value of some process p_k. As a first step, let us assume p_i decided the value v_i because p_i received a broadcast message b that told p_i to decide, and let us assume that this is the only way to make p_i decide. Now, a proof without history variables would require going back in time to the point where p_i obtained the message b. Hence, a lemma would be needed showing that if p_i decides in run R, there must be one and only one point in time in R before the decision is made, when p_i got the broadcast message etc. In our model, in most cases we can omit reasoning along the timeline because the information needed is still there, although if it is 'older' than the inspected configuration. A *received* tag will be set for received broadcast messages but the messages are still part of the configuration. Hence, we can formulate an (configuration) invariant: 'Whenever the decision value of a process p_i is set to v_i, then there is a broadcast message with content v_i'. Hence, all that remains to be shown is that all broadcast messages contain only input values. Let us assume that broadcast messages' contents are generated from the information in point to point messages. Since these messages are also still available, it remains to establish an invariant that states that all messages only contain input values and so on. The developed libraries for interprocess communication will be presented in Chapter 4 and more detailed characteristics of proofs for the inspected algorithms are given in Chapter 5.

3.4. Utilizing Fairness Assumptions

If we did not assume that there is any progress in a distributed system, it would be impossible to show liveness properties like Termination. In other words, we are only able to prove properties like Termination if we assume that processes are not permitted to remain idle. Hence, we need some *fairness* assumptions forcing the constituent parts to make progress if progress is possible.

Considering our introduced model, there is already an inherent notion of fairness in our runs. By the definition of a run, the algorithm is only allowed to do nothing if we end up in a deadlock.

Note that this is a fundamental difference to models like e.g. TLA$^+$ where the system is supposed to have the option of *stuttering* (repeat a state/configuration) for an infinitely long time.

Despite this intrinsic fairness, for some algorithms more assumptions have to be added to guarantee the desired properties. Moreover, [FLP85] showed that without further assumptions it is not possible to solve Consensus in asynchronous environments with crash failures. Due to the unbounded delay of messages, in an asynchronous system, processes are not aware if other processes have already crashed or just need more time to answer to a posed request.

In [CT96] failure detectors are proposed to overcome the lack of synchrony and make Consensus solvable. As explained in Section 2.3.8, failure detectors can be seen as modules located at the processes. For a process p_i, the local failure detector module provides a list of processes that are presumed to be crashed. This helps processes decide whether to wait for delayed messages or not. If the information provided were to be totally correct, the assumption would be stronger than needed. Hence, the delivered list of crashed processes is assumed to be erroneous and the question examined in [CT96] is, how incorrect the list may be without making Consensus unsolvable, in other words the task is to find the weakest assumptions for the failure detector under which Consensus is still solvable. There may be of course further assumptions that are needed as e.g.:

- Bounds on the maximum number of processes that crash during a run.
- Timing assumptions for maximum delay times of messages sent and for process speed.
- Assumptions on the message infrastructure as e.g. that every sent message is eventually received.
- Fairness assumptions for the modeled actions as Weak or Strong Fairness (c.f. [Lam02]).

Our definition of a distributed algorithm $\mathfrak{A} = (\text{ Init}, \; \Phi, \; \Psi, \; \mathcal{F}_{\text{ad}} \;)$ supports additive assumptions by the component $\mathcal{F}_{\text{ad}}$. As an example assume there is a function crashed : $\mathbb{C} \rightarrow \mathcal{P}(\mathbb{P})$ that returns the set of crashed processes in a given configuration, then the predicate

$$\text{maxCrash}_n : \Sigma \rightarrow \mathbf{bool} \mid \text{maxCrash}_n(s) \triangleq \forall t \in \text{dom}(s) \; . \; |\mathit{crashed}(s(t))| \leq n$$

is true if and only if there are n processes crashed at maximum in a sequence s.

Hence, by equipping an algorithm with $\mathcal{F}_{\text{ad}} = \{ \text{ maxCrash}_n \}$ we obtain Admissible Runs where less (or equal) than n processes crash. Of course, it is much more difficult to implement timing assumptions in our model. In Chapter 9 of [Lam02] Leslie Lamport illustrates how real time can be implemented in TLA$^+$ models. The same techniques can be applied to our model. In Chapter 5 we will show further applications of $\mathcal{F}_{\text{ad}}$.

Regarding our proofs for Termination, the scheme for proving liveness properties can be depicted as follows: At first we show that, due to the assumed fairness assumptions, every correct process will reach a certain point in the execution, where, from the external global view, we know it has reached a decision. Then, by contradiction we assume that there is a correct process p_i that does not decide and show that, on the other hand p_i will reach the point where it must have decided. This implies the contradiction. Obviously, the challenge is to identify the point in time that must have been reached by every correct process. Most effort will be spent to show that every p_i will indeed reach this point. But since there are infinitely many possible fairness assumptions which vary widely, we are not able to develop a universal proof methodology. Practical examples will be given in Chapter 5.

4. Distributed Algorithms in Isabelle/HOL

Our main goal is to establish a methodology that enables us to formally model and verify a given distributed algorithm within a theorem proving environment. This methodology of course should be applicable for different distributed algorithms and hence we focus on the reusability of all single parts of our model and verification techniques. This section presents our library that can be seen as a tool box for modeling and verification of distributed algorithms in Isabelle/HOL.

To have a reusable approach for different algorithms the concept of encapsulating a distributed algorithm in a locale is borrowed from [CDM11]. Locales for Isabelle are used to section theories and can be used as a simple form of modules [KWP99]. Theorems proven abstractly for the locale can then be used for every instantiation of the locale. Hence theorems proven for an abstract locale *`DistributedAlgorithm`* can be used in every concrete instantiation of this locale. An instantiation of a locale in Isabelle is called ***`interpretation`***.

This work uses a locale *`DistributedAlgorithm`* and accordingly every distributed algorithm considered in the case study will be an ***`interpretation`*** of this locale or of some extension of this locale. The different algorithms are thereby defined by passing parameters to the locale. These parameters correspond to the entries of our tuple representation of a distributed algorithm (cf. Chapter 2).

Of course the locale *`DistributedAlgorithm`* is central to our formalization. But without communication a distributed system would just be a collection of sequential processes. Therefore, beside the models of concrete algorithms that will be presented in Chapter 5 and the presented locale, a further essential part of our formalization is the interprocess communication (IPC). A distributed system may use different options for communication, e.g. message passing, shared memory, and remote procedure calls. All in all we developed four theories containing definitions, datastructures and theorems that generally assist with modelling processes, process interactions, and with proving the correctness of the aggregated distributed system:

1. A theory containing the basic definitions concerning the set of processes ($\hookrightarrow$ Isabelle[1]).

2. A theory containing the presented locale *`DistributedAlgorithm`* and additional definitions and lemmas to work with the locale ($\hookrightarrow$ Isabelle[2]). The theory with the locale uses the afore mentioned basic definitions (see 1.).

3. Theories featuring the communication via message passing. If communication is supposed to work over messages these theories can be applied. In the following sections we will present different versions that have been developed to support different message passing

[1]Isabelle/HOL theory: `DistrAlgBasic.thy`
[2]Isabelle/HOL theory: `DistributedAlgorithm.thy`

infrastructures. ($\hookrightarrow$ Isabelle[3], $\hookrightarrow$ Isabelle[4] and $\hookrightarrow$ Isabelle[5])

4. A theory for shared memory, i.e. for regular registers (cf. Section 2.3.7, see [Lam85]). Although Regular Registers are only a certain kind of shared memory, many typical problems and characteristics are enlightened by this example. ($\hookrightarrow$ Isabelle[6])

To model a newly developed distributed algorithm with our library, an interpretation of our locale (defined in 2) must be written. The interpretation has to define the parameters for the locale. Defining actions and events requires to formalize the communication. According to requirements, the respective communication theory (from 3. or 4.) must be chosen. The following sections explain the details of the four introduced theories. The descriptions are given by the annotated Isabelle libraries.

4.1. Basic Definitions

theory *DistrAlgBasic* **imports** *Main*
begin

At first we define the time domain $\mathbb{T}$ to be the set of natural numbers.

— we define runs on a discrete timeline
type_synonym *T = nat*

We axiomatize that the constant *N* with *1* $\leq$ *N* represents the (fixed) number of processes of the considered distributed system.

axiomatization
  N — the count of procs in our Model
  where
  — There will be at least one process
  NgreaterZero: *"1≤(N::nat)"*

Furthermore, we define a type *proc* for the processes. We define *proc* by turning the set $\{ n \in \mathbb{N} \mid 1 \leq n \leq N \}$ into a new type. The correspondent set of natural numbers can be interpreted as the set of process ids (PIDs).

typedef *proc = "{(1::nat)..N}"*

definition
  proc :: "nat set" **where**
  "proc ≡ {(1::nat)..N}"

The set of processes is now defined as the set of all *i* of type *proc*

definition

[3]Isabelle/HOL theory: `MsgPassBasic.thy`
[4]Isabelle/HOL theory: `MsgPassNoLoss.thy`
[5]Isabelle/HOL theory: `MsgPassWLoss.thy`
[6]Isabelle/HOL theory: `Register.thy`

```
procs :: "proc set" where
"procs ≡ {i::proc. True}"
```

Sometimes it will be helpful to order procs by their process id. Hence we define an ordering on the processes. The function `Rep_proc p` returns the natural number that corresponds to the process `p` (based on the definition of the type `proc`) and hence is used as the process id of process `p`.

```
abbreviation "PID ≡ Rep_proc"
```

The ordering is then simply applied from the ordering of natural numbers.

```
instantiation "proc" :: linorder
begin
```

Hence, we define `p1 < p2` (`p1 ≤ p2`) if and only if the process id of `p1` is smaller (smaller or equal) than the process id of `p2`.

```
definition le_proc_def: "p1 ≤ p2 ≡ PID p1 ≤ PID p2"
definition less_proc_def: "p1 < p2 ≡ PID p1 < PID p2"
instance
end
```

Our first three lemmas concern the finiteness of sets of processes and process ids:

1. The set of process ids is finite.
2. The set of processes `procs` is finite.
3. Since there are only `N` different processes, every arbitrary set of processes must be finite.

Note that the notation `PID ‘ procs` denotes the set of process ids that are obtained by the application of `PID` for every element of `procs`. Hence

$$\mathit{PID}\ ‘\ \mathit{procs} \triangleq \{\, x \mid \exists y \in \mathit{procs} \,.\, \mathit{PID}\ y = x \,\}$$

```
lemma FinitePIDs: shows "finite (PID ‘ procs)"

lemma FiniteProcs: shows "finite procs"

lemma FiniteSubsetProcs: shows "finite (A::proc set)"
```

The function `Abs_proc` denotes the inverse function of `PID`, hence, it returns processes to given process ids. Since `proc` is the set of natural numbers from 1 to `N`, `Abs_proc ‘ proc` is the set of processes `procs`.

```
lemma ProcsAbs: "procs = Abs_proc ‘ proc"
```

Finally, we deduce that we have `N` elements in `procs`.

```
lemma ProcsCard: "card procs = N"

end
```

4.2. Distributed Algorithm

```
theory DistributedAlgorithm
imports DistrAlgBasic
begin
```

This theory contains the fundamental definitions for our concept of algorithms and distributed algorithm as it has been introduced in Section 2.3. This includes:

- Definitions for the types of localviews, configurations, actions etc.
- The definitions of the locales *SafeAlgorithm*, *Algorithm*, and *DistributedAlgorithm*, which conform to the definitions of algorithms (cf. Definition 2.3.1.3, Definition 2.3.1.8 and Definition 2.3.2.3)
- Definitions for steps, deadlocks (cf. Definition 2.3.1.4), and runs (cf. Definition 2.3.1.6)
- Definitions for component preserving process-actions, and events (cf. Definition 3.1.0.6)
- Definitions for enabledness (cf. Definition 2.3.1.10)

Moreover, it contains helpful theorems for reasoning over modeled algorithms where an algorithm is an interpretation of the locale.

At first, we define the basic type *conf* for a configuration. This type is modeled as a record and already contains a field *P_State* for the array of process states (this corresponds to **S**, cf. Section 2.3.1). The type of process-states depends on the algorithm and hence on the interpretation of the locale. Therefore, we use a type variable *'ps* for the type of the process-state.

```
record 'ps conf  =
  P_State :: "proc ⇒ 'ps"
```

To be able to extend the record for a configuration by further modules, we have to use the scheme *('ps, 'a) conf_scheme* of the defined record *'ps conf* (see Section 8.2.2 in [NPW02]). The extension requires to use a further type variable *'a*. Therefore, the type we employ uses two type variables (*'ps*,*'a*). We define a type synonym to abbreviate *('ps, 'a) conf_scheme*.

```
type_synonym ('ps,'a) confT =  "(('ps,'a) conf_scheme)"
```

In the following *('ps, 'a) confT* will be used as the type for configurations.

The type of a process-action is a predicate over configuration pairs and processes. We introduce a type *procActionT* for process-actions:

```
type_synonym
('ps, 'a) procActionT = "((('ps,'a) confT) ⇒ (('ps,'a) confT) ⇒ proc ⇒ bool)"
```

Events are predicates over configurations pairs. We introduce a type *eventT* for events:

```
type_synonym ('ps, 'a) eventT = "((('ps,'a) confT) ⇒ (('ps,'a) confT) ⇒ bool)"
```

Note that the type variables for configurations (*'ps* and *'a*) are also required for actions and events because the type of actions and events depend on the type of configurations.

4.2.1. Safe Algorithms and Algorithms

Now we are ready to introduce the locale for Safe Algorithms (cf. Definition 2.3.1.3). As described in Section 2.3.1 a Safe Algorithm is completely defined by the set of valid initial configurations and the actions and events of the algorithm. Therefore, our locale has three parameters:

- *InitPred*: A predicate that defines the initial configurations of the algorithm. In our model this corresponds to the Init predicate of a distributed algorithm $\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}}\ )$.
- *ProcActionSet*: A (finite) set of Actions that can be performed by processes. In our model this corresponds to the set Φ of a distributed algorithm $\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}}\ )$.
- *EventSet*: A (finite) set of Events. In our model this corresponds to the set Ψ of a distributed algorithm $\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi,\ \mathcal{F}_{\text{ad}}\ )$.

The definition of a Safe Algorithm requires the sets of actions and events to be finite.

```
locale SafeAlgorithm =
  fixes
  InitPred      :: "(('ps,'a) confT) ⇒ bool" and
  ProcActionSet :: "(('ps,'a) procActionT) set" and
  EventSet      :: "(('ps,'a) eventT) set"
  assumes
  ActionSetsFin:
  "finite ProcActionSet"
  "finite EventSet"
```

context *SafeAlgorithm* **begin**

In the context of the locale *SafeAlgorithm*, we consider an arbitrary fixed Safe Algorithm $\mathfrak{A} = (\text{ Init},\ \Phi,\ \Psi\ )$ (in terms of our Isabelle notation: (*InitPred*, *ProcActionSet*, *EventSet*)). Hence the variables Init, Φ, and Ψ (which correspond to *InitPred*,*ProcActionSet*,*EventSet*in the Isabelle formalization) are fixed variables in the following definitions, lemmas and theorems.

As introduced in Section 2.3.1, computation steps of our model generate discrete transitions from one configuration to the next. As already explained in Section 2.3.1, these steps are caused by defined actions. Actions can be performed by processes, e.g., if a process manipulates its state or sends a message the system transits from one state to the next. In most environments there are also things that occur without any influence of a process. As an example consider transitions where a message is lost due to the instable communication media. Therefore, we distinguish the introduced types of actions in our Isabelle-model:

- process-actions - process-actions are performed by processes and hence are ternary relations between two configurations and one process. If $\mathcal{A}_p(\mathfrak{C}, \mathfrak{C}', p_i)$ for two configurations $\mathfrak{C}, \mathfrak{C}'$ and a process p_i holds then this can be interpreted as p_i executes $\mathcal{A}_p$ in configuration c and the result is a transition to configuration c' (note that in Section 2.3.1 we introduced the notation $\mathfrak{C} \rightarrow_{p_i:\mathcal{A}} \mathfrak{C}'$ for $\mathcal{A}_p(\mathfrak{C}, \mathfrak{C}', p_i)$). Since actions are not always deterministic, there might be $\mathfrak{C}'' \neq \mathfrak{C}'$ such that $\mathcal{A}_p(\mathfrak{C}, \mathfrak{C}', p_i)$ and $\mathcal{A}_p(\mathfrak{C}, \mathfrak{C}'', p_i)$.

- events - events are performed independently from the processes. If $\mathcal{A}$ is an event holding for two configurations $\mathfrak{C}, \mathfrak{C}'$, $\mathcal{A}$ happens in configuration $\mathfrak{C}$ and the system enters configuration $\mathfrak{C}'$.

These types of actions induce two defined types of steps (cf. Section 2.3.1).

- *ProcessStep*s - Steps directly caused by processes, i.e. executions of ProcActions.
- *Event*s - Steps that are not caused by a certain process, i.e. occurences of events where no dedicated process is responsible for the transition to the next configuration.

A *Step* of the system is then defined as the happening of a process-step or an event-step (cf. Definition 2.3.1.4).

```
definition
  ProcessStep:: "((’ps,’a) confT) ⇒ ((’ps,’a) confT) ⇒ bool" where
  "ProcessStep c c’ ≡ ∃i. ∃A ∈ ProcActionSet. A c c’ i"
```

```
definition
  Event:: "((’ps,’a) confT) ⇒ ((’ps,’a) confT) ⇒ bool" where
  "Event c c’ ≡ ∃A ∈ EventSet. A c c’"
```

```
definition
  Step:: "((’ps,’a) confT) ⇒ ((’ps,’a) confT) ⇒ bool" (infixl "→" 900) where
  "Step c c’ ≡ ProcessStep c c’ ∨ Event c c’"
```

If the system reaches a configuration where no further step is possible, the system is deadlocked (cf. Definition 2.3.1.4).

```
definition
  deadlock :: "((’ps,’a) confT) ⇒ bool" where
  "deadlock c ≡ ∀c’. ¬ c → c’"
```

Verification for an algorithm $\mathfrak{A}$ requires to make assertions about all possible executions of $\mathfrak{A}$. As explained in Section 2.3.1, a run is an infinite sequence of configurations where *InitPred* holds in the initial configuration and every configuration and its successor is in the *Step* relation (cf. Definition 2.3.1.6). Of course, deadlocks may occur (configurations where no further steps are possible). Due to the type restrictions of Isabelle, the option of having finite *Runs* is avoided and therefore in these cases *stuttering* is allowed (see Section 2.3.1, cf. [Lam02]), i.e. to infinitely repeat the deadlocked configuration. Therefore, we define a predicate *FinalStuttering* that allows to stutter if the system is deadlocked.

```
definition
  FinalStuttering :: "((’ps,’a) confT) ⇒ ((’ps,’a) confT) ⇒ bool" where
  "FinalStuttering s s’ ≡ (s = s’) ∧ deadlock s"
```

```
definition
  Run :: "(T ⇒ ((’ps,’a) confT)) ⇒ bool" where
  "Run R ≡ InitPred (R 0) ∧ (∀t::T. ((R t) → (R (t+1)))
  ∨ FinalStuttering (R t) (R (t+1)))"
```

An *InfiniteRun* is a run without deadlocks.

```
definition
  InfiniteRun :: "(T ⇒ (('ps,'a) confT)) ⇒ bool" where
  "InfiniteRun R ≡ Run R ∧ (∀t. ¬ deadlock (R t) )"
```

In the following, we show two basic lemmas for deadlocks. The first states that if a run R exhibits a deadlock in a configuration $R(t)$ then configuration $R(t+1)$ is also deadlocked and we have $R(t) = R(t+1)$ (the step from $R(t)$ to $R(t+1)$ is a finalstuttering step):

```
lemma deadlockLastsForeverStep:
  assumes R: "Run R" and tm: "deadlock (R t)"
  shows "deadlock (R (Suc t))" "(R t) = (R (Suc t))"
```

Using the previous lemma, we can now show Proposition *Deadlocks last forever* (Proposition 2.3.1.7) from Section 2.3.1.

```
lemma deadlockLastsForever:
  assumes R: "Run R" and tm: "deadlock (R t)" and t: "t' ≥ t"
  shows "deadlock (R t')" "(R t) = (R t')"
```

For process-actions, we define a predicate *Enabled_A* to determine whether it is possible to execute an action in a given configuration by a given process (cf. Definition 2.3.1.10). For events, we define the respective predicate *Enabled_E*. This corresponds to Definition 2.3.1.10.

```
definition
  Enabled_A
  :: "(('ps,'a) procActionT)
  ⇒ (('ps,'a) confT) ⇒ proc
  ⇒ bool" where
  "Enabled_A A c i ≡ ∃c'. A c c' i"

definition
  Enabled_E ::
  "(('ps,'a) eventT) ⇒ (('ps,'a) confT) ⇒ bool" where
  "Enabled_E A c ≡ ∃c'. A c c'"
```

For actions and events we define predicates *ComponentPreservingA* and *ComponentPreservingE*. As described in Definition 3.1.0.6, an action preserves a component if the component is not changed by the action. For a given component *cp* the set *Ξ_A cp* (respectively *Ξ_E cp*) contains all process-actions (respectively events) which are not component preserving for *cp*.

```
definition
  ComponentPreservingA ::
  "((('ps,'a) confT) ⇒ (('ps,'a) confT) ⇒ proc ⇒ bool)
  ⇒ ((('ps,'a) confT) ⇒ 'xyz)
  ⇒ bool" where
  "ComponentPreservingA A cp ≡ ∀i. ∀c c'. A c c' i ⟶ cp c = cp c'"

definition
```

```
  ComponentPreservingE ::
  "((('ps,'a) confT) ⇒ (('ps,'a) confT) ⇒ bool)
  ⇒ ((('ps,'a) confT) ⇒ 'xyz)
  ⇒ bool" where
  "ComponentPreservingE A cp ≡ ∀c c'. A c c' ⟶ cp c = cp c'"

definition
  Ξ_A
  :: "((('ps,'a) confT) ⇒ 'xyz)
  ⇒ ((('ps,'a) confT)  ⇒ (('ps,'a) confT)  ⇒ proc ⇒ bool) set" where
  "Ξ_A cp ≡ {A ∈ ProcActionSet. ¬ ComponentPreservingA A cp}"

definition
  Ξ_E
  :: "((('ps,'a) confT) ⇒ 'xyz)
  ⇒ ((('ps,'a) confT) ⇒ (('ps,'a) confT) ⇒ bool) set" where
  "Ξ_E cp ≡ {A ∈ EventSet. ¬ ComponentPreservingE A cp}"
```

As a basic lemma we show that whenever a configuration $R(t)$ differs from its successor $R(t+1)$ in a run R

- there is no deadlock in $R(t)$
- there is either a process-step or an event-step from $R(t)$ to $R(t+1)$.

This lemma is fundamental for reasoning methods as described in Sections 3.1 and 3.2.

```
lemma ConfigInv: assumes R: "Run R" and Rneq: "R t ≠ R (Suc t)"
  shows "¬ deadlock (R t)" "ProcessStep (R t) (R (Suc t))∨ Event (R t) (R (Suc t))"
```

By the definition of component preserving events(process-steps) it is easy to show that if there is an event(process-step) between two configurations $\mathfrak{C}$ and $\mathfrak{C}'$ that preserves component *cp*, then cp($\mathfrak{C}$) = cp($\mathfrak{C}$').

```
lemma NotConcernedC:
  assumes A: "ComponentPreservingE A cp" and step: "A c c'"
  shows "cp c = cp c'"

lemma NotConcernedL:
  assumes A: "ComponentPreservingA A cp" and step: "A c c' i"
  shows "cp c = cp c'"
```

If there is a process-step between two configurations $\mathfrak{C}$ and $\mathfrak{C}'$ and a component *cp* is not preserved (i.e. $cp(\mathfrak{C}) \neq cp(\mathfrak{C}')$), then (of course) there must be a process p_i and a process-action $\mathcal{A}$ such that $\mathcal{A}$ is executed by p_i from $\mathfrak{C}$ to $\mathfrak{C}'$ and $\mathcal{A}$ does not preserve component *cp*. Therefore, we can obtain a fixed process-action $\mathcal{A} \in \Phi$ and a process $p_i \in \mathbb{P}$ such that $\mathcal{A}$ is one of the actions in Φ that do not preserve component *cp* and $\mathfrak{C} \rightarrow_{p_i:\mathcal{A}} \mathfrak{C}'$.

```
lemma GetProcessAction:
  assumes step: "ProcessStep c c'" and cp: "cp c ≠ cp c'"
  obtains A i where "A ∈ ProcActionSet" "¬ ComponentPreservingA A cp" "A c c' i"
```

The respective lemma for events assumes an event between two configurations $\mathfrak{C}$ and $\mathfrak{C}'$ and obtains a fixed $\mathcal{A} \in \Psi$ that does not preserve component *cp*.

```
lemma GetEvent:
  assumes step: "Event c c'" and cp: "cp c ≠ cp c'"
  obtains A where "A ∈ EventSet" "¬ ComponentPreservingE A cp" "A c c'"
```

Using the introduced lemmas and definitions, we can now show Theorem 3.1.0.7 which helps to reduce the cases to consider in invariant-based reasoning (cf. Section 3.1):

```
theorem NonCompPreservingActionsMatter:
  assumes R: "Run R"
  and x1: "∀A ∈ (Ξ_A cp). ∀i.
                A (R t) (R (Suc t)) i ∧ P(cp(R t)) ⟶ P(cp(R (Suc t)))"
  and x2: "∀A ∈ (Ξ_E cp).
                A (R t) (R (Suc t)) ∧ P(cp(R t)) ⟶ P(cp(R (Suc t)))"
  shows "P(cp(R t)) ⟶ P(cp(R (Suc t)))"
```

To show an invariant P by induction, we can use the following lemma, which states that if we assume the invariant holds in $R(t)$ and every step preserves the invariant, it still holds in $R(t+1)$. Finalstuttering steps do not need to be considered, since they imply $R(t) = R(t+1)$ and therefore $P(R(t)) = P(R(t+1))$.

```
lemma FinalStutteringInd:
  assumes R: "Run R"
  and P: "P (R t)"
  and st: "((R t) → (R (t+1))) ⟹ P (R (Suc t))"
  shows "P (R (Suc t))"
```

Based on the definitions of component preserving actions and events we formulate another induction Theorem `CPInduct`, which can be used to show invariants and which has been already discussed in Section 3.2 (cf. Theorem 3.2.0.11). To prove that $P(cp(R(t)))$ holds for every arbitrary t, we have to show $P\,(cp\,(R\,(0)))$ for the base case. For the step we assume $P\,(cp\,(R\,(t)))$ for induction hypothesis and have to show that every step that does not preserve *cp* implies $P\,(cp\,(R\,(t+1)))$.

```
theorem CPInduct: assumes R: "Run R"
  and IH: "P (cp (R 0))"
  and IS: "⋀t' i A.
              ⟦A ∈ Ξ_A cp; A (R t') (R (Suc t')) i;P(cp(R t'))⟧
                                           ⟹ P (cp (R (Suc t')))"
  "⋀t' A.
              ⟦A ∈ Ξ_E cp; A (R t') (R (Suc t'));P(cp(R t'))⟧
                                           ⟹ P (cp (R (Suc t')))"
shows "P(cp(R t))"
```

The following theorem can be used to show assertions of the kind of Irrevocability, i.e., propositions that assert if a condition $b(c(R(t)))$ is satisfied for a component c at time t then for all $t' \geq t$ we have $c(R(t)) = c(R(t+1))$ (cf. Theorem 3.2.0.12).

```
theorem CondGe:
  assumes step: "⋀t'. ⟦b(c((s::nat⇒(('ps,'a) confT))(t')))⟧
                                          ⟹ c(s(t')) = c(s(t'+1))"
  and b: "b(c(s(t)))"
  shows "∀t' ≥ t. c(s(t)) = c(s(t'))"
```

end — Safe Algorithm context

Now we use the definition of a Safe Algorithm to define an Algorithm (cf. Section 2.3.1). For an Algorithm we extend the definition by a parameter $\mathcal{F}_{\text{ad}}$, which corresponds to `AdmissiblePreds` in our Isabelle formalization. $\mathcal{F}_{\text{ad}}$ must be a set of predicates that define the Admissible Runs of the algorithm (cf. Definition 2.3.1.9).

```
locale Algorithm = SafeAlgorithm +
  fixes
  AdmissiblePreds:: "((T ⇒ (('a,'b) confT)) ⇒ bool) set"
```

context *Algorithm* **begin**

An `AdmissibleRun` is a run where additionally the predicates in set `AdmissiblePreds` hold (which corresponds to $\mathcal{F}_{\text{ad}}$ in Section 2.3.1).

```
definition
  AdmissibleRun :: "(T ⇒ (('a,'b) confT)) ⇒ bool" where
  "AdmissibleRun R ≡ Run R ∧ (∀P ∈ AdmissiblePreds. P R)"
```

end — Algorithm context

4.2.2. Distributed Algorithms

We define a basic type `AbsLocalView` for a Localview to be used in our definition of a Distributed Algorithm. This type is modeled as a record and already contains a field `P_State_i` for process states (this corresponds to **S**, cf. Section 2.3.2). The type of process-states depends on the algorithm and hence on the interpretation of the locale. Therefore, we use a type variable `'ps` for the type of the process-state.

```
record 'ps AbsLocalView =
  P_State_i :: "'ps"
```

To be able to extend the record for a Localview by further entries, we have to use the scheme `('ps, 'a) conf_scheme` of the defined record `'ps AbsLocalView` (see Section 8.2.2 in [NPW02]). The extension requires to use a further type variable `'b`. Therefore the type we employ uses two type variables (`'ps`,`'b`). We define a type synonym to abbreviate `('ps, 'b) AbsLocalView_scheme`:

type_synonym

```
('ps,'b) LocViewT =  "(('ps,'b) AbsLocalView_scheme)"
```

The type of a non-lifted action is $\mathbb{L} \times \mathbb{L} \rightarrow$ **bool**. In Isabelle we define a type synonym for the type of non-lifted actions:

type_synonym
```
('ps, 'b) nonliftedactionT =  "(('ps,'b) LocViewT) ⇒ (('ps,'b) LocViewT) ⇒ bool"
```

The introduced function Lift depends on the definitions of C2LV and LV2C for the respective Distributed Algorithm (cf. Definition 2.3.2.2). Therefore in Isabelle we define a parametrized version `ParamLift_A` which takes these functions and a non-lifted action as arguments.

definition
```
  ParamLift_A :: "
  ((('ps,'a) confT) ⇒ proc ⇒ ('ps,'b) LocViewT)
  ⇒ (('ps,'b) LocViewT ⇒ proc ⇒ (('ps,'a) confT) ⇒ (('ps,'a) confT))
  ⇒ (('ps,'b) nonliftedactionT)
  ⇒ ('ps,'a) procActionT" where
  "ParamLift_A Conf2LocView LocView2Conf a ≡ (λc. λc'. λi::proc. ∃lv.
  a (Conf2LocView c i) lv
  ∧ c' = LocView2Conf lv i c)"
```

For the definition of the type of a DnA, we use a record based on a type variable `'lv` for the type of a Localview.

record `'lv DnA =`
```
  enabledcond :: "'lv ⇒ bool"
  smallstep   :: "'lv ⇒ 'lv"
```

Analogously to the definition of `ParamLift_A`, we define a parametrized version of DLift that takes C2LV, LV2C, and a DnA as arguments and returns a corresponding process-action.

definition
```
  ParamDLift_A :: "
  ((('ps,'a) confT) ⇒ proc ⇒ ('ps,'b) LocViewT)
  ⇒ (('ps,'b) LocViewT ⇒ proc ⇒ (('ps,'a) confT) ⇒ (('ps,'a) confT))
  ⇒(('ps,'b) LocViewT) DnA
  ⇒ (('ps,'a) procActionT)" where
  "ParamDLift_A Conf2LocView LocView2Conf a ≡
      (λc.λc'.λi. (enabledcond a) (Conf2LocView c i)
               ∧ c' = (LocView2Conf ((smallstep a) (Conf2LocView c i)) i c))"
```

As introduced in Section 2.3.2, we define a predicate `LocalViewRespect`, which is true if an process-action respects the Localview of a process (cf. LocalviewRespect). `LocalViewRespect` also depends on functions C2LV and LV2C and therefore they must be given as arguments.

definition
```
  LocalViewRespect :: "
  ((('ps,'a) confT) ⇒ proc ⇒ ('ps,'b) LocViewT)
  ⇒ (('ps,'b) LocViewT ⇒ proc ⇒ (('ps,'a) confT) ⇒ (('ps,'a) confT))
  ⇒ (('ps,'a) procActionT) ⇒ bool" where
  "LocalViewRespect Conf2LocView LocView2Conf A ≡
                              ∃a. A = ParamLift_A Conf2LocView LocView2Conf a"
```

We show that every lifted DnA satisfies *LocalViewRespect* for arbitrary functions *C2LV* and *LV2C* (by the definition of *LocalViewRespect* this is equivalent to Theorem 2.3.3.3).

```
lemma DLiftImpLocalViewRespect1:
  "LocalViewRespect C2LV LV2C (ParamDLift_A C2LV LV2C a)"
```

Let *ap* be a set of DnAs. By the previous lemma we can deduce that every process-action *A* in the set of the lifted actions of *ap* satisfies *LocalViewRespect*.

```
theorem DLiftImpLocalViewRespect2: fixes
  "Conf2LocView"::"
            ((('ps,'a) confT) ⇒ proc ⇒ ('ps,'b) LocViewT)" and
  "LocView2Conf"::
          "(('ps,'b) LocViewT ⇒ proc ⇒ (('ps,'a) confT) ⇒ (('ps,'a) confT))"
  assumes a: "A ∈ ((ParamDLift_A Conf2LocView LocView2Conf) ` ap)"
  shows "LocalViewRespect Conf2LocView LocView2Conf A"
```

As explained in Section 2.3.1, we use a predicate LocallyEqual to formulate the assertion NoRemoteAccess. Two configurations are LocallyEqual for a process $i \in \mathbb{P}$ if and only if the process-state of i and the extension (the states of the further modules) are equal in both configurations.

```
definition
  LocallyEqual :: "(('ps,'a) confT) ⇒ (('ps,'a) confT) ⇒ proc ⇒ bool" where
  "LocallyEqual c1 c2 i ≡ P_State c1 i = P_State c2 i ∧ conf.more c1 = conf.more c2"
```

Now we are able to define the locale for a Distributed Algorithm. For the definition we extend the defined algorithm by the parameters for C2LV and LV2C. Furthermore, we postulate that for a Distributed Algorithm the constraints locViewProp1, locViewProp2, LocalviewRespect, NoRemoteAccess and StateInvEv must be satisfied (cf. definition2.3.2.3). Therefore, we add the assumptions *locViewProp1*, *locViewProp2*, *LocalViewRespect*, *NoRemoteAccess* and *StateInvEv* to our locale.

```
locale DistributedAlgorithm = Algorithm +
  fixes
  Conf2LocView   :: "(('a,'b) confT) ⇒ proc ⇒ ('a,'lvt) LocViewT" and
  LocView2Conf   :: "('a,'lvt) LocViewT ⇒ proc ⇒ (('a,'b) confT) ⇒ (('a,'b) confT)"
  assumes
  StateInvEv:
  "⋀c c' A. ⟦A∈EventSet; A c c' ⟧⟹ P_State c = P_State c'" and
  locViewProp1:
  "LocView2Conf (Conf2LocView c i) i c = c" and
  locViewProp2:
  "P_State (LocView2Conf lv i c) = (P_State c) (i := P_State_i lv)" and
  LocalViewRespect:
  "⋀A. ⟦A ∈ ProcActionSet⟧ ⟹ LocalViewRespect Conf2LocView LocView2Conf A" and
  NoRemoteAccess:
  "⋀c c' A i c1.
  ⟦A∈ProcActionSet; A c c' i; LocallyEqual c c1 i⟧
```

```
⟹ A c1 ⦇P_State = (P_State c1)(i := P_State c' i), ... = conf.more c'⦈ i"
```

For every interpretation of this locale it has to be shown that the action sets are finite and the assertions locViewProp1, locViewProp2, LocalviewRespect, NoRemoteAccess, and StateInvEv hold for the concrete instances of the parameters.

This models a distributed algorithm as introduced in Chapter 2. Starting from the defined initial configurations, the execution of the algorithm corresponds to the successive execution of actions from the given action sets.

context *DistributedAlgorithm* **begin**

In the context of the locale *DistributedAlgorithm*, we consider an arbitrary fixed Distributed Algorithm $\mathfrak{A} = (\text{ Init}, \ \Phi, \ \Psi, \ \mathcal{F}_{\text{ad}}, \ \text{C2LV}, \ \text{LV2C })$.

For $\mathfrak{A}$ we can deduce from locViewProp1 and locViewProp2 that for a process p_i and a configuration $\mathfrak{C}$ we have $\mathbf{S}_{\mathfrak{C}}(p_i) = \mathbf{S}_{\text{C2LV}(\mathfrak{C},p_i)}$.

lemma *locViewProp0:* **shows** *"P_State_i (Conf2LocView c i) = P_State c i"*

By StateInvEv we have that events do not manipulate the process states. By LocalviewRespect we can deduce that processes do only manipulate their own process state if they execute a process-action but not the process state of other processes.

```
lemma StateInv:
  "⋀c c' i j A. ⟦A∈ProcActionSet; A c c' i; P_State c j ≠ P_State c' j⟧⟹ i = j"
  "⋀c c' A. ⟦A∈EventSet; A c c' ⟧⟹ P_State c = P_State c'"
```

Therefore, if there is a process $p_j \neq p_i$ and p_i executes a process-action $\mathcal{A}$ from a configuration $\mathfrak{C}$ to a configuration $\mathfrak{C}'$, then $\mathbf{S}_{\mathfrak{C}}(p_j) = \mathbf{S}_{\mathfrak{C}'}(p_j)$.

```
lemma StateNotConcerned:
  assumes A: "A ∈ ProcActionSet"
      and step: "A c c' i"
      and i: "i ≠ j"
  shows "P_State c j = P_State c' j"
```

Since events do not manipulate the process states, every $\mathcal{A} \in \Psi$ preserves the component $\mathbf{S}$ (note that $\mathbf{S}$ can be interpreted as a component since it maps configurations to an array of process states).

```
lemma EventspreserveStates:
  assumes R: "Run R" and A: "A ∈ EventSet"
  shows "ComponentPreservingE A P_State"
```

Furthermore, since events do not manipulate process states, we can deduce that if we have $\mathbf{S}_{R(t)} \neq \mathbf{S}_{R(t+1)}$ there must be a process-action $\mathcal{A}$ and a process p_i such that $R(t) \rightarrow_{p_i:\mathcal{A}} R(t+1)$. Hence, we can obtain the respective action and process.

```
lemma GetAction:
  assumes R: "Run R"
  and st: "P_State (R t) i ≠ P_State (R (Suc t)) i"
  obtains A where "A ∈ ProcActionSet" "A (R t) (R (Suc t)) i"
```

As described in Section 3.1, for showing invariants that are assertions on process states we do not need to consider events but only process-actions in the respective case distinction (cf. Corollary 3.1.0.9).

```
corollary NonStPreservingPAMatter:
  assumes R: "Run R"
  and x1: "∀A ∈ (Ξ_A P_State). ∀i.
                A (R t) (R (Suc t)) i ∧ P(P_State(R t)) ⟶ P(P_State(R (Suc t)))"
  shows "P(P_State(R t)) ⟶ P(P_State(R (Suc t)))"
```

As stated in Section 3.1, if we have an assertion about the state of p_i, it suffices to consider the cases in which p_i executes a $process - action$ (cf. Theorem 3.1.0.10).

```
corollary NonStPreservingPAMatter2:
  assumes R: "Run R" and
  x1: "∀A ∈ (Ξ_A P_State).
          A (R t) (R (Suc t)) i ∧ P((P_State(R t))i) ⟶ P(P_State((R (Suc t)))i)"
  shows "P((P_State(R t))i) ⟶ P((P_State(R (Suc t)))i)"

end — Distributed Algorithm context

end — theory
```

4.3. Message Passing

```
theory MsgPassBasic
imports DistrAlgBasic
begin
```

This theory implements the basic concepts introduced in Section 2.3.5. It provides the fundamental functions to keep track of messages by using a Message History module. To be used by a distributed algorithm the type of a configuration must be extended by an entry for the Message History. A basic type for this entry is defined in this theory but the type actually used depends on the type of messages the algorithm defines. A basic message type `Message` is introduced as a record in this module. It should be extended by entries for the content of messages. The basic type only provides the entries `snd` and `rcv` for the sender and the receiver of a message.

```
record Message =
  snd :: proc
  rcv :: proc
```

As described in Section 2.3.5, a Message History maps messages to message status values. Since we assume there might be multiple copies of a message in the system, the message status counts the number of copies for each possible status. As introduced in Section 2.3.5, the basic options for the status of a message are given by the set $\mathcal{T}_{\text{ags}}$= { **outgoing**, **transit**, **received** }. These basic options will be extended later by the theory for message loss (cf. Section 2.3.5, Message passing with message loss).

```
record MsgStatus =
  outgoing :: nat
  transit  :: nat
  received :: nat
```

Using the types `Message` and `MsgStatus`, we define a type synonym for the type of a Message History. As already explained, a Message History is a mapping from messages to message status values. Since both types `Message` and `MsgStatus` are supposed to be extended, we have to use the schemes `'a Message_scheme` and `'exMS MsgStatus_scheme` of the defined records (see Section 8.2.2 in [NPW02]). Therefore, the resulting type synonym makes use of two type variables: `'a` for the extension of a message and `'exMS` for the extension of the message status.

```
type_synonym
  ('a,'exMS) msghistoryT =  "('a Message_scheme) ⇒ ('exMS MsgStatus_scheme)"
```

To add an **outgoing** message, we define a function `incOutgoing`, which takes a message status and increments the value for the outgoing copies and returns a modified message status.

```
definition
  incOutgoing :: "('exMS MsgStatus_scheme) ⇒ ('exMS MsgStatus_scheme)" where
  "incOutgoing st ≡ st ⦇outgoing := Suc (outgoing st)⦈"
```

Furthermore, we define the functions outMsgs, transitmsgs, and recmsgs as introduced in Definition 2.3.5.1 in Isabelle as `OutgoingMsgs`,`TransitMsgs` and `ReceivedMsgs`.

```
definition
  OutgoingMsgs :: "(('a,'exMS) msghistoryT) ⇒ (('a Message_scheme) set)" where
  "OutgoingMsgs M ≡ {m. outgoing (M m) > 0}"

definition
  TransitMsgs :: "(('a,'exMS) msghistoryT) ⇒ (('a Message_scheme) set)" where
  "TransitMsgs M ≡ {m. transit (M m) > 0}"

definition
  ReceivedMsgs :: "(('a,'exMS) msghistoryT) ⇒ (('a Message_scheme) set)" where
  "ReceivedMsgs M ≡ {m. received (M m) > 0}"
```

To send a set of messages, we introduced the subaction MultiSend in Section 2.3.5. This subaction returns true for two Message Histories **Q** and **Q**′ and a set of messages *msgs* if **Q**′ is an update of **Q** such that each **outgoing** value of each message of the set *msgs* is incremented by one in **Q**′.

```
definition
  MultiSend :: "
  (('a,'exMS) msghistoryT)
  ⇒ (('a,'exMS) msghistoryT)
  ⇒ ('a Message_scheme) set
  ⇒ bool" where
```

```
"MultiSend Q Q' msgs ≡ Q' = (λm. if (m ∈ msgs) then incOutgoing (Q m) else Q m)"
```

For the use with DnAs, we provide a functional version `MultiSend_Get` of `MultiSend`. For a given Message History **Q** and a set of messages *msgs*, `MultiSend_Get` returns the updated Message History **Q′**, where each value of **Q′** is equal to the value of **Q** but the **outgoing** values of the messages in *msgs* are incremented by one.

```
definition
  MultiSend_Get ::
  "(('a,'exMS) msghistoryT)
  ⇒ (('a Message_scheme) set)
  ⇒ (('a,'exMS) msghistoryT)" where
  "MultiSend_Get Q msgs ≡ (λm. if (m ∈ msgs) then incOutgoing (Q m) else Q m)"
```

In the following, the subactions Send, Transmit, Receive, and Duplicate are defined as introduced in Section 2.3.5:

- `Send` increments the **outgoing** value of a single message.
- `Transmit` decrements the **outgoing** and increments the **transit** value of a given message.
- `Receive` decrements the **transit** and increments the **received** value of a given message.
- `Duplicate` creates another copy of a message in **transit** by incrementing the **transit** value.

Note that these subactions are only enabled if the values that are decremented from **Q** to **Q′** are greater than zero. Furthermore, the execution of `Duplicate` requires that there must be at least one message that can be copied, i.e. the value **transit** of the given message must be greater than zero.

```
definition
  Send ::
  "(('a,'exMS) msghistoryT)
  ⇒ (('a,'exMS) msghistoryT)
  ⇒ ('a Message_scheme)
  ⇒ bool" where
  "Send Q Q' m ≡ Q' = Q (m := (incOutgoing (Q m)))"

definition
  Transmit ::
  "(('a,'exMS) msghistoryT)
  ⇒ (('a,'exMS) msghistoryT)
  ⇒ ('a Message_scheme)
  ⇒ bool" where
  "Transmit Q Q' m ≡ m ∈ OutgoingMsgs Q
  ∧ Q' = Q (m := (Q m)
          ⦇ outgoing := (outgoing (Q m))-(1::nat),
            transit := Suc(transit (Q m))⦈)"

definition
  Receive ::
```

```
  "(('a,'exMS) msghistoryT)
  ⇒ (('a,'exMS) msghistoryT)
  ⇒ ('a Message_scheme)
  ⇒ bool" where
  "Receive Q Q' m ≡ m ∈ TransitMsgs Q
  ∧ Q' = Q (m := (Q m)
        (| transit := transit (Q m)-(1::nat),
           received := Suc(received (Q m)) |))"
```

```
definition
  Duplicate ::
  "(('a,'exMS) msghistoryT)
  ⇒ (('a,'exMS) msghistoryT)
  ⇒ ('a Message_scheme)
  ⇒ bool" where
  "Duplicate Q Q' m ≡ m ∈ TransitMsgs Q
  ∧ Q' = Q (m := (Q m)(| transit := Suc(transit (Q m)) |))"

end — theory
```

4.3.1. Message Passing without Message Loss

```
theory MsgPassNoLoss
imports MsgPassBasic
begin
```

This theory extends the basic theory for message passing for the case where there is no message loss and therefore the message status record needs no further entries.

In this case, for a Message History **Q** the set of messages msgs(**Q**) (cf. Section 2.3.5) is the set of messages where for each $m \in$ msgs(**Q**) at least one of the values **outgoing**, **transit**, or **received** is greater than zero.

```
definition Msgs ::
  "(('a Message_scheme)
  ⇒ MsgStatus)
  ⇒ (('a Message_scheme) set)" where
  "Msgs M ≡ {m. outgoing (M m) > 0 ∨ transit (M m) > 0 ∨ received (M m) > 0}"
```

Initially there are no messages or copies of messages in the system. Hence, we assume that every initial Message History must return zero values for the **outgoing**, **transit**, and **received** values of every message. Therefore, we define the initial Message History as follows:

```
definition InitialMsgHistory :: "('a Message_scheme) ⇒ MsgStatus" where
  "InitialMsgHistory ≡ (λm. (|outgoing = 0, transit = 0, received = 0|))"
```

In Section 2.3.5 we introduced functions SumReceived$_{\mathbf{Q}}$, SumTransit$_{\mathbf{Q}}$, and SumOutgoing$_{\mathbf{Q}}$ to determine the corresponding numbers of copies of messages that are **received**, in **transit** and **outgoing** in a Message History **Q**. In the following, these functions are defined in Isabelle as *RemReceived*, *RemTransit*, and *RemOutgoing*.

```
definition RemReceived :: "(('a Message_scheme) ⇒ MsgStatus) ⇒ nat" where
  "RemReceived Q = (∑m ∈ Msgs Q. received (Q m))"

definition RemTransit :: "(('a Message_scheme) ⇒ MsgStatus) ⇒ nat" where
  "RemTransit Q = (∑m ∈ Msgs Q. transit (Q m))"

definition RemOutgoing :: "(('a Message_scheme) ⇒ MsgStatus) ⇒ nat" where
  "RemOutgoing Q = (∑m ∈ Msgs Q. outgoing (Q m))"
```

Finally, the sum of all messages in a Message History **Q** is retrieved by adding the return values of all three defined functions. Hence, the corresponding function for $\text{SumMsgs}_{\mathbf{Q}}$ is defined as follows:

```
definition SumMessages :: "(('a Message_scheme) ⇒ MsgStatus) ⇒ nat" where
  "SumMessages Q = (RemReceived Q + RemTransit Q + RemOutgoing Q)"
```

As explained in Section 2.3.5, we use a function MaxRemMsgMoves to calculate how many transitions due to the movement of messages in a Message History **Q** are possible. This function is here given by `MaxRemMsgMoves`:

```
definition MaxRemMsgMoves :: "(('a Message_scheme) ⇒ MsgStatus) ⇒ nat" where
  "MaxRemMsgMoves Q = RemTransit Q + (2*RemOutgoing Q)"
```

Sometimes it is easier to use alternative definitions for $\text{SumReceived}_{\mathbf{Q}}$, $\text{SumTransit}_{\mathbf{Q}}$, and $\text{SumOutgoing}_{\mathbf{Q}}$ in the proofs. Therefore, the following lemmas provide alternative representations where summation is only made over the respective $\text{recmsgs}_{\mathbf{Q}}$, $\text{transitmsgs}_{\mathbf{Q}}$, and $\text{outMsgs}_{\mathbf{Q}}$ sets.

```
lemma RemReceivedAlt:
  assumes fin: "finite (Msgs Q)"
  shows "RemReceived Q = (∑m ∈ ReceivedMsgs Q. received (Q m))"

lemma RemTransitAlt:
  assumes fin: "finite (Msgs Q)"
  shows "RemTransit Q = (∑m ∈ TransitMsgs Q. transit (Q m))"

lemma RemOutgoingAlt:
  assumes fin: "finite (Msgs Q)"
  shows "RemOutgoing Q = (∑m ∈ OutgoingMsgs Q. outgoing (Q m))"
```

As explained in Section 2.3.5, for the different message sets we can deduce some simple properties like subset relations etc. (cf. Proposition 2.3.5.3, Proposition 2.3.5.4, Proposition A.0.0.2, Proposition A.0.0.3, and Proposition A.0.0.4). Therefore, the following lemmas support convenient reasoning over those message sets by listing up simple facts, which are retrieved by using the definitions.

```
lemma MsgsProps:
  shows
  "OutgoingMsgs c ⊆ Msgs c"
  "TransitMsgs c ⊆ Msgs c"
  "ReceivedMsgs c ⊆ Msgs c"
  "OutgoingMsgs c ∪ TransitMsgs c ∪ ReceivedMsgs c = Msgs c"

lemma MultiSendProps:
  assumes x:
  "MultiSend c c' msgs"
  shows
  "OutgoingMsgs c' = OutgoingMsgs c ∪ msgs"
  "TransitMsgs c' = TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c"

lemma MultiSendMsgChange:
  assumes x: "MultiSend c c' msgs"
  shows
  "Msgs c ⊆ Msgs c'"
  "Msgs c' = Msgs c ∪ msgs"

lemma TransmitProps:
  assumes x: "Transmit c c' m"
  shows
  "OutgoingMsgs c' ∪ {m} = OutgoingMsgs c"
  "OutgoingMsgs c' ⊆ OutgoingMsgs c"
  "TransitMsgs c' = TransitMsgs c ∪ {m}"
  "ReceivedMsgs c' = ReceivedMsgs c"

lemma ReceiveProps:
  assumes x: "Receive c c' m"
  shows
  "TransitMsgs c' ∪ {m} = TransitMsgs c"
  "OutgoingMsgs c' = OutgoingMsgs c"
  "TransitMsgs c' ⊆ TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c ∪ {m}"

lemma SendProps:
  assumes x: "Send c c' m"
  shows
  "OutgoingMsgs c' = OutgoingMsgs c ∪ {m}"
  "OutgoingMsgs c ⊆ OutgoingMsgs c'"
  "TransitMsgs c' = TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c"

lemma DuplicateProps:
  assumes x: "Duplicate c c' m"
  shows
  "OutgoingMsgs c' = OutgoingMsgs c"
  "TransitMsgs c' = TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c"
  "Msgs c = Msgs c'"
```

```
lemma SendMsgChange:
  assumes x: "Send c c' m"
  shows
  "Msgs c ⊆ Msgs c'"
  "Msgs c' = Msgs c ∪ {m}"
```

Since subactions Receive, and Transmit only shift messages between the different subsets of $\text{msgs}_{\mathbf{Q}}$, the execution of such a subaction does not change $\text{msgs}_{\mathbf{Q}}$. This is expressed by the lemma *MsgsInvariant* (cf. Proposition 2.3.5.5).

```
lemma MsgsInvariant:
  shows
  "Receive c c' m ⟹ Msgs c' = Msgs c"
  "Transmit c c' m ⟹ Msgs c' = Msgs c"
```

The following lemma describes how a Transmit step affects SumOutgoing, SumTransit, and SumReceived (cf. Lemma 2.3.5.6).

```
lemma TransmitInv:
  assumes fin: "finite (Msgs Q)"
  and tm: "Transmit Q Q' m"
  shows
  "RemOutgoing Q > 0"
  "RemTransit Q' = Suc (RemTransit Q)"
  "RemOutgoing (Q') = RemOutgoing (Q)-(1::nat)"
  "RemReceived Q' = RemReceived Q"
```

Analogously to the previous lemma, the following lemma describes how a Receive step affects SumOutgoing, SumTransit, and SumReceived (cf. Lemma 2.3.5.7).

```
lemma ReceiveInv:
  assumes fin: "finite (Msgs Q)"
  and tm: "Receive Q Q' m"
  shows
  "RemTransit Q > 0"
  "RemOutgoing Q' = RemOutgoing Q"
  "RemTransit (Q') = RemTransit (Q)-(1::nat)"
  "RemReceived Q' = Suc (RemReceived Q)"
```

By the previous two lemmas we can deduce that the sum of messages is invariant under the execution of the subactions Transmit, and Receive(cf. Theorem 2.3.5.8).

```
lemma SumInv:
  assumes fin: "finite (Msgs Q)"
  shows
  "Transmit Q Q' m ⟹ SumMessages Q = SumMessages Q'"
  "Receive Q Q' m ⟹ SumMessages Q = SumMessages Q'"
```

end — theory

4.3.2. Message Passing with Message Loss

theory *MsgPassWLoss*
imports *MsgPassBasic*
begin

This theory extends our basic theory for message passing for the case in which message loss is possible during the transmission of the message. Therefore, the message status record needs to be extended by an entry for the lost messages (cf. Section 2.3.5).

```
record MsgStatusWLoss = MsgStatus +
  lost     :: nat
```

Hence, a Message History is a mapping from message schemes to the new type *MsgStatusWLoss*.

```
type_synonym
  'a msghistoryT =  "('a Message_scheme) ⇒ MsgStatusWLoss"
```

As for the case with no message loss, initially there are no messages or copies of messages in the system. Hence, we assume that every initial Message History must return zero values for the **outgoing**, **transit**, **received** and also for the **lost** values of every message. Therefore, we define the initial Message History as follows:

```
definition InitialMsgHistory :: "'a msghistoryT" where
  "InitialMsgHistory ≡ (λm. (|outgoing = 0, transit = 0, received = 0, lost = 0|))"
```

In this case, for a Message History **C** the set of messages msgs(**C**) (cf. Section 2.3.5) is the set of messages where for each $m \in$ msgs(**C**) at least one of the values **outgoing**, **transit**, **received** or **lost** is greater than zero.

```
definition Msgs ::
  "('a msghistoryT) ⇒ (('a Message_scheme) set)" where
  "Msgs C ≡ {m. outgoing (C m) > 0
             ∨ transit (C m)  > 0
             ∨ received (C m) > 0
             ∨ lost (C m)     > 0}"
```

In addition to the sets $\text{outMsgs}_{\mathbf{C}}$, $\text{transitmsgs}_{\mathbf{C}}$ and $\text{recmsgs}_{\mathbf{C}}$ that have been defined in *MsgPassBasic* we define the set $\text{lostmsgs}_{\mathbf{C}}$ for the messages that have been lost (cf. Section 2.3.5).

```
definition LostMsgs :: "('a msghistoryT) ⇒ (('a Message_scheme) set)" where
  "LostMsgs C ≡ {m. lost (C m) > 0}"
```

Furthermore, in addition to the subactions MultiSend, Send, Transmit, and Receive, we provide a subaction Lose to describe how a message gets lost (cf. Section 2.3.5). For a loss of a message we decrement the **transit** and increment the **lost** value of a message.

```
definition Lose ::
  "('a msghistoryT)
  ⇒ ('a msghistoryT)
  ⇒ ('a Message_scheme)
  ⇒ bool" where
  "Lose C C' m ≡ m ∈ TransitMsgs C
  ∧ C' = C (m := (C m)⦇ transit := transit (C m)-(1::nat), lost := Suc(lost (C m))⦈)"
```

As for the case without message loss, for the different message sets we can deduce some simple properties like subset relations etc. (cf. Proposition 2.3.5.3, Proposition 2.3.5.4, Proposition A.0.0.2, Proposition A.0.0.3 and Proposition A.0.0.4). Therefore, the following lemmas support convenient reasoning over those message sets by listing up simple facts, which are retrieved by using the definitions. The respective lemmas are very similar to the case without message loss but are adapted for the set lostmsgs $_{\mathbf{C}}$ and for the subaction Lose.

```
lemma MsgsProps:
  shows
  "OutgoingMsgs c ⊆ Msgs c"
  "OutgoingMsgs c ⊆ Msgs c"
  "TransitMsgs c ⊆ Msgs c"
  "ReceivedMsgs c ⊆ Msgs c"
  "LostMsgs c ⊆ Msgs c"
  "OutgoingMsgs c ∪ TransitMsgs c ∪ ReceivedMsgs c ∪ LostMsgs c = Msgs c"

lemma MultiSendProps:
  assumes x: "MultiSend c c' msgs"
  shows
  "OutgoingMsgs c' = OutgoingMsgs c ∪ msgs"
  "TransitMsgs c' = TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c"
  "LostMsgs c' = LostMsgs c"

lemma MultiSendMsgChange:
  assumes x: "MultiSend c c' msgs"
  shows
  "Msgs c ⊆ Msgs c'"
  "Msgs c' = Msgs c ∪ msgs"

lemma SendMsgChange:
  assumes x: "Send c c' m"
  shows
  "Msgs c ⊆ Msgs c'"
  "Msgs c' = Msgs c ∪ {m}"

lemma SendProps:
  assumes x: "Send c c' m"
  shows
  "OutgoingMsgs c' = OutgoingMsgs c ∪ {m}"
  "OutgoingMsgs c ⊆ OutgoingMsgs c'"
  "TransitMsgs c' = TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c"
  "LostMsgs c' = LostMsgs c"


lemma TransmitProps:
  assumes x: "Transmit c c' m"
  shows
  "OutgoingMsgs c' ∪ {m} = OutgoingMsgs c"
  "OutgoingMsgs c' ⊆ OutgoingMsgs c"
  "TransitMsgs c' = TransitMsgs c ∪ {m} "
  "ReceivedMsgs c' = ReceivedMsgs c"
  "LostMsgs c' = LostMsgs c"
  "Msgs c = Msgs c'"

lemma ReceiveProps:
  assumes x: "Receive c c' m"
  shows
  "TransitMsgs c' ∪ {m} = TransitMsgs c"
  "OutgoingMsgs c' = OutgoingMsgs c"
  "TransitMsgs c' ⊆ TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c ∪ {m}"
  "LostMsgs c' = LostMsgs c"
  "Msgs c = Msgs c'"

lemma DuplicateProps:
  assumes x: "Duplicate c c' m"
  shows
  "OutgoingMsgs c' = OutgoingMsgs c"
  "TransitMsgs c = TransitMsgs c'"
  "ReceivedMsgs c' = ReceivedMsgs c"
  "LostMsgs c' = LostMsgs c"
  "Msgs c = Msgs c'"

lemma LoseProps:
  assumes x: "Lose c c' m"
  shows
  "OutgoingMsgs c' = OutgoingMsgs c"
  "TransitMsgs c' ⊆ TransitMsgs c"
  "ReceivedMsgs c' = ReceivedMsgs c"
  "LostMsgs c' = LostMsgs c ∪ {m}"
  "Msgs c = Msgs c'"

end — theory
```

4.4. Regular Registers

```
theory Register
imports DistributedAlgorithm
begin
```

This theory provides datastructures, definitions, and theorems for modeling Distributed Algorithms that use shared memory, i.e., the Regular Register abstraction, which was introduced by Lamport (cf. [Lam85]). Since Lamport's abstractions are action-based, whereas our model is state-based, in Section 2.3.7 we introduced a new state-based model for Regular Registers. A main contribution of this theory is a proof which shows that both models are equivalent. The main ideas of this model and of the proof have already been discussed in Section 2.3.7. Therefore, in the following we will focus on the specific characteristics of the Isabelle formalization and only give references to the definitions, theorems, and explanations of Section 2.3.7.

For $\mathcal{V}_{\text{reg}}$ we use a type variable *'content* which represents the possible content of a register. As introduced in Section 2.3.7 processes make use of a read-oriented and a write-oriented view on the registers in our model. As explained, during the transient phase of writing, the value view^{w} of the write-oriented view is a pair $(v_{old}, v_{new}) \in (\mathcal{V}_{\text{reg}} \times \mathcal{V}_{\text{reg}})$. For the stable phases between two write operations, the value of view^{w} is a value $v \in \mathcal{V}_{\text{reg}}$. In Isabelle, we model this by a datatype *reg_state*:

```
datatype 'content reg_state =
   WriteInProgress "'content" "'content"
 | WriteStable "'content"
```

Therefore, a variable of type *'content reg_state* can either be *WriteInProgress vold vnew* or *WriteStable v*, where *vold*,*vnew* and *v* are arbitrary values of type *'content*.

The introduced function vset is therefore defined as follows (in Isabelle we use the name *regValueSet*):

```
definition regValueSet
  :: "'content reg_state ⇒ 'content set" where
  "regValueSet reg ≡ (case reg of
  (WriteStable cnt) ⇒ {cnt}
  | (WriteInProgress cnt cnt') ⇒ {cnt,cnt'})"
```

For a variable *vieww* of type *'content reg_state*, it returns a set V, where V contains only *cnt* if *vieww = WriteStable cnt* and if *vieww = WriteInProgress cnt cnt'* then $V =$ *{cnt, cnt'}*. It is easy to see that for a given *x*, *regValueSet x* can never be the empty set.

```
lemma regValueNotEmpty: shows "regValueSet x ≠ {}"
```

We use another datatype to model the different phases of $\mathcal{R}_{phs}$:

```
datatype RegPhase = RIdle | RReading "proc" | RWriting
```

Note that we use a parameter to store which register a process reads. Since there is only one register that a process is allowed to write, this parameter is not needed for the writing phase.

As described in Section 2.3.7, the register status of a process p_i is a triple with three entries:

- the register phase of p_i
- p_i's write-oriented view on its own register Reg_{p_i}
- p_i's read-oriented view on all registers

In Isabelle, we make use of a record to easen the access to the single entries.

```
record 'content ProcRegStatus =
  regphs :: RegPhase
  wReg :: "'content reg_state"
  rRegs:: "proc ⇒ 'content set"
```

The type of a Shared Memory History is then defined as a mapping from $\mathbb{P}$ to the set of register status values:

```
type_synonym 'content shmemhistoryT = "proc ⇒ 'content ProcRegStatus"
```

4.4.1. Subactions and Atoms

Now we define the different subactions as they have defined and explained in Section 2.3.7. We start with the subactions for the begin and end of a write operation, followed by the begin and end of a read operation. Details, explanations and examples for applications are given in Section 2.3.7 (cf. subactions *WriteBegin*, *WriteEnd*, *RdBegin*, *RdEnd*).

```
— For the begin of a write operation
definition
  RegWriteBegin ::
  "('content shmemhistoryT)
  ⇒ ('content shmemhistoryT)
  ⇒ proc
  ⇒ 'content
  ⇒ bool" where
  "RegWriteBegin c c' i cnt' ≡
  regphs (c i) = RIdle
  ∧ (∃cnt. wReg (c i) = WriteStable cnt
  ∧ c' = (λj. (|
  regphs = (if (i = j) then RWriting else regphs (c j)),
  wReg = (if (i = j) then WriteInProgress cnt cnt' else wReg (c j)),
  rRegs= (rRegs (c j)) ( i:= rRegs (c j) i ∪ {cnt'})
  |)))"

— For the end of a write operation
definition
  RegWriteEnd ::
  "('content shmemhistoryT)
  ⇒ ('content shmemhistoryT)
  ⇒ proc
  ⇒ bool" where
  "RegWriteEnd c c' i ≡
  regphs (c i) = RWriting
  ∧ (∃cnt cnt'. wReg (c i) = WriteInProgress cnt cnt'
  ∧ c' = (c) (i := ⦇ regphs = RIdle, wReg = WriteStable cnt', rRegs = rRegs (c i)⦈))"
```

— For the begin of a read operation
definition

```
  RegReadBegin ::
  "('content shmemhistoryT)
  ⇒ ('content shmemhistoryT)
  ⇒ proc
  ⇒ proc
  ⇒ bool" where
  "RegReadBegin c c' i regno ≡
  regphs (c i) = RIdle
  ∧ c' = (c) (i := ⦇ regphs = RReading regno,
                     wReg   = wReg (c i),
                     rRegs = (rRegs (c i))
                             (regno := regValueSet (wReg (c regno)))
                   ⦈)"
```

— For the end of a read operation
definition RegReadEnd ::

```
  "('content shmemhistoryT)
  ⇒ ('content shmemhistoryT)
  ⇒ proc
  ⇒ proc
  ⇒ 'content
  ⇒ bool" where
  "RegReadEnd c c' i regno cnt ≡
  regphs (c i) = RReading regno
  ∧ cnt ∈ ((rRegs (c i)) regno)
  ∧ c' = (c) (i := ⦇ regphs = RIdle, wReg = wReg (c i), rRegs = (rRegs (c i))⦈)"
```

As explained in Section 2.3.7, we also provide atoms for the use in DnAs. As described for each subaction a two atoms a_{en} (for the use in the enabled predicate of a DnA) and $a_{\triangle}$ (for the use in the transition function of a DnA) are defined (cf. $WriteBegin_{en}$, $WriteBegin_{\triangle}$, $WriteEnd_{en}$, $WriteEnd_{\triangle}$, $RdBegin_{en}$, $RdBegin_{\triangle}$, $RdEnd_{en}$, $RdEnd_{\triangle}$). In Isabelle we denote a_{en} atoms by `a_Enabled` and the $a_{\triangle}$ atoms by `a_Delta`.

— Atoms for `RegWriteBegin`
definition

```
  RegWriteBegin_Enabled ::
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ bool" where
  "RegWriteBegin_Enabled c i ≡  regphs (c i) = RIdle
                              ∧ (∃cnt. wReg (c i) = WriteStable cnt)"
```

definition

```
  RegWriteBegin_Delta ::
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ 'content
  ⇒ ('content shmemhistoryT)" where
  "RegWriteBegin_Delta c i cnt' ≡
              (λj. let cnt = (ϵ cnt. wReg (c i) = WriteStable cnt) in
                 (| regphs = (if (i = j) then RWriting else regphs (c j)),
                    wReg   = (if (i = j) then
                                     WriteInProgress cnt cnt'
                               else
                                     wReg (c j)),
                    rRegs  = (rRegs (c j)) ( i:= rRegs (c j) i ∪ {cnt'})
                 |))"
```

— Atoms for *RegWriteEnd*
definition

```
  Enabled_RegWriteEnd ::
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ bool" where
  "Enabled_RegWriteEnd c i ≡ regphs (c i) = RWriting
  ∧ (∃ cnt cnt'. wReg (c i) = WriteInProgress cnt cnt')"
```

definition

```
  RegWriteEnd_Delta ::
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ ('content shmemhistoryT)" where
  "RegWriteEnd_Delta c i ≡
          let cnt' = (ϵ cnt'. ∃ cnt. wReg (c i) = WriteInProgress cnt cnt') in
                ((c) (i := (c i)(| regphs := RIdle, wReg := WriteStable cnt'|)))"
```

— Atoms for *RegReadBegin*
definition *Enabled_RegReadBegin ::*

```
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ bool" where
  "Enabled_RegReadBegin c i ≡ regphs (c i) = RIdle"
```

definition *RegReadBegin_Delta ::*

```
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ proc
  ⇒ ('content shmemhistoryT)" where
  "RegReadBegin_Delta c i regno ≡
                          (c) (i := (| regphs = RReading regno,
                                       wReg   = wReg (c i),
                                       rRegs  = (rRegs (c i))
                                           (regno := regValueSet (wReg (c regno)))
```

```
                                          |))"
```

— Atoms for *RegReadEnd*

```
definition
  Enabled_RegReadEnd ::
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ proc
  ⇒ 'content
  ⇒ bool" where
  "Enabled_RegReadEnd c i regno cnt ≡
  regphs (c i) = RReading regno
  ∧ cnt ∈ ((rRegs (c i)) regno)"

definition
  RegReadEnd_Delta ::
  "('content shmemhistoryT)
  ⇒ proc
  ⇒ ('content shmemhistoryT)" where
  "RegReadEnd_Delta c i ≡ (c) (i :=
                    (| regphs = RIdle,
                       wReg   = wReg (c i),
                       rRegs  = (rRegs (c i))
                    |))"
```

By the following theorems, we prove that the subactions *WriteBegin*, *WriteEnd*, *RdBegin*, and *RdEnd* can be constructed from the respective atoms (cf. Theorem 2.3.7.5). In order to prove the main result, for every subaction `a` we show that (⇒) if the enabled predicate `a_enabled` holds for a Shared Memory History `c` and a process `i` and the Shared Memory History `c'` equals `a_Delta c i`, then `a c c' i` holds and (⇐) vice versa (note that for some cases more parameters are required for the atoms and the subactions). Theorems `RWBEqEnExecRWB`, `RWEEqEnExecRWE`, `RRBEqEnExecRRB` and `RREEqEnExecRRE` constitute Theorem 2.3.7.5. For each theorem we prove (⇒) and (⇐).

```
theorem RWBEqEnExecRWB:
  "RegWriteBegin c c' i (cnt'::'content) = (RegWriteBegin_Enabled c i
                                          ∧ c' = RegWriteBegin_Delta c i cnt')"
theorem RWEEqEnExecRWE:
  "RegWriteEnd c c' i = (Enabled_RegWriteEnd c i
                       ∧ c' = RegWriteEnd_Delta c i)"
theorem RRBEqEnExecRRB:
  "RegReadBegin c c' i regno = (Enabled_RegReadBegin c i
                              ∧ c' = RegReadBegin_Delta c i regno)"
theorem RREEqEnExecRRE:
  "RegReadEnd c c' i regno cnt = (Enabled_RegReadEnd c i regno cnt
                                ∧ c' = RegReadEnd_Delta c i)"
```

4.4.2. Regular Registers used for Distributed Algorithms

To separate valid steps from steps that access the shared memory in an uninteded way, we define the notion of ProperMemSteps (cf. Definition 2.3.7.1).

```
definition
  ProperMemStep ::
  "('content shmemhistoryT)
  ⇒ ('content shmemhistoryT)
  ⇒ proc
  ⇒ bool" where
  "ProperMemStep c c' i ≡ (∃cnt'. RegWriteBegin c c' i cnt')
  ∨ (RegWriteEnd c c' i)
  ∨ (∃regno. RegReadBegin c c' i regno)
  ∨ (∃regno cnt. RegReadEnd c c' i regno cnt)"
```

As explained in Section 2.3.7, we assume a mapping initval : $\mathbb{P} \to \mathcal{V}_{\text{reg}}$ such that initval(p_j) returns the initial value of a register Reg_{p_j}. For the initial Shared Memory History, all register phases must be **rIdle**, all write-oriented views on registers must be set to the respective initial values, and all read-oriented views must be set to the empty set. Note that in our Isabelle formalization `InitialValue` corresponds to initval in our formal model.

```
definition InitReg ::
  "(proc ⇒ 'content)
  ⇒ ('content shmemhistoryT)
  ⇒ bool" where
  "InitReg InitialValue rc ≡ ∀i. regphs (rc i) = RIdle
  ∧ wReg (rc i) = WriteStable (InitialValue i)
  ∧ (∀j. rRegs (rc i) j = {})"
```

To use our Shared Memory History module, we provide another locale SMAlgorithm, which extends the locale `DistributedAlgorithm`. It needs two further parameters:

- `SM`: A mapping that returns the Shared Memory History for a given configuration (hence in our terms we would define $\mathit{SM}(\mathfrak{C}) = \mathbf{Sh}_{\mathfrak{C}}$.
- `InitialValue`: The mapping that defines the initial values of the registers.

```
locale SMAlgorithm = DistributedAlgorithm +
  fixes SM :: "(('a, 'b) conf_scheme) ⇒ ('content shmemhistoryT)"
  fixes InitialValue :: "proc ⇒ 'content"
  assumes
  ProperSteps: "⋀A i c c'. ⟦A∈ProcActionSet; A c c' i; SM c ≠ SM c'⟧
                              ⟹ ProperMemStep (SM c) (SM c') i"
               "⋀A c c'. ⟦A∈EventSet; A c c'⟧⟹ SM c = SM c'"
  and InitRegPred: "InitPred c ⟹ InitReg InitialValue (SM c)"
```

The assumptions of the locale guarantee that every process-step of the algorithm is either a ProperMemStep or it does not change the Shared Memory and that all initial configurations are configurations where the correspondend initial values are stored in the registers.

context *SMAlgorithm* **begin**

In the context of the locale SMAlgorithm, we consider an arbitrary fixed algorithm with the additional parameters. At first we introduce some definitions to be used in the context of our locale.

The predicate *existsNextPMStep* is true for a run R, a process p_i and a time t if and only if there is a time $t' \geq t$ such that there is a ProperMemStep by p_i in t'.

```
definition
  existsNextPMStep ::
  "(T ⇒ (('a, 'b) conf_scheme))
  ⇒ proc
  ⇒ T
  ⇒ bool" where
  "existsNextPMStep R i t ≡ ∃t' ≥ t. ProperMemStep (SM (R t')) (SM (R (Suc t'))) i"
```

Analogously, the predicate *existsLastPMStep* is true for a run R, a process p_i and a time t if and only if there is a time $t' < t$ such that there is a ProperMemStep by p_i in t'.

```
definition
  existsLastPMStep ::
  "(T ⇒ (('a, 'b) conf_scheme))
  ⇒ proc
  ⇒ T
  ⇒ bool" where
  "existsLastPMStep R i t ≡ ∃t' < t. ProperMemStep (SM (R t')) (SM (R (Suc t'))) i"
```

We define a function *GetNextPMStep* which for a run R, a process p_i and a time t returns the minimum time t' such that $t' \geq t$ and there is a ProperMemStep of p_i in t'. We use an option datatype, which allows to return a value *None* for special cases (see Section 1.4.2 and [NPW02]). Here it is used to return *None* in case there is no next ProperMemStep.

```
definition GetNextPMStep ::
  "(T ⇒ (('a, 'b) conf_scheme))
  ⇒ proc
  ⇒ T
  ⇒ T option" where
  "GetNextPMStep R i t ≡ if ¬ existsNextPMStep R i t then
                          None
                         else
                          Some (ϵt'. t' ≥ t
                           ∧ ProperMemStep (SM (R t')) (SM (R (Suc t'))) i
                           ∧ (∀t''. ProperMemStep (SM (R t'')) (SM (R (Suc t''))) i
                                 ∧ t'' ≥ t ∧ t'' ≠ t' ⟶ t'' > t'))"
```

Analogously, we define a function *GetLastPMStep* which for a run R, a process p_i and a time t returns the maximum time t' such that $t' < t$ and there is a ProperMemStep of p_i in t'. Again an option datatype is used such that *None* is returned in case there is no last ProperMemStep.

```
definition GetLastPMStep ::
  "(T ⇒ (('a, 'b) conf_scheme))
  ⇒ proc
  ⇒ T
  ⇒ T option" where
"GetLastPMStep R i t ≡ if ¬ existsLastPMStep R i t then None
  else
  Some (Max {t'. t' < t ∧ ProperMemStep (SM (R t')) (SM (R (Suc t'))) i})"
```

Now we can show some basic properties for the definitions made. The first lemma states: if there is a next ProperMemStep for a run R, process p_i and a time t then for R, p_i and t

- *GetNextPMStep* returns not *None*
- *GetNextPMStep* returns a time $t' \geq t$
- there is a ProperMemStep at the time *GetNextPMStep*
- there are no ProperMemSteps of p_i between t and the time *GetNextPMStep* returns

```
lemma GetNextPMStepProps:
  assumes ex: "existsNextPMStep R i t"
  shows
  "GetNextPMStep R i t ≠ None"
  "the(GetNextPMStep R i t) ≥ t"
  "ProperMemStep
          (SM (R (the(GetNextPMStep R i t))))
          (SM (R (Suc (the(GetNextPMStep R i t)))))
          i"
  "(∀t''. ProperMemStep (SM (R t'')) (SM (R (Suc t''))) i
        ∧ t'' ≥ t
        ∧ t'' ≠ the(GetNextPMStep R i t)
     ⟶ t'' > the(GetNextPMStep R i t))"
```

A dual result is proved for *GetLastPMStep*. We show that: if there is a last ProperMemStep for a run R, process p_i and a time t then for R, p_i and t

- *GetNextPMStep* returns not *None*
- *GetNextPMStep* returns a time $t' < t$
- there is a ProperMemStep at the time *GetNextPMStep*
- there are no ProperMemSteps of p_i between the time *GetLastPMStep* returns and t

```
lemma GetLastPMStepProps: assumes ex: "existsLastPMStep R i t"
  shows
  "GetLastPMStep R i t ≠ None"
  "the(GetLastPMStep R i t) < t"
  "ProperMemStep
            (SM (R (the(GetLastPMStep R i t))))
            (SM (R (Suc (the(GetLastPMStep R i t)))))
            i"
  "(∀t''. ProperMemStep (SM (R t'')) (SM (R (Suc t''))) i
        ∧ t'' < t
        ∧ t'' ≠ the(GetLastPMStep R i t)
     ⟶ t'' < the(GetLastPMStep R i t))"
```

By the previous theorems we can deduce that for a run R, a process p_i and for time t if we have a time t' with $t' \geq t$ and *GetNextPMStep* returns *None* or a value greater than t' for R, p_i and t, then there is no ProperMemStep of p_i at t' in R.

```
lemma NoProperMemStepInBetween1: assumes R: "Run R"
  and t': "t' ≥ t"
  and nn: "t' < the(GetNextPMStep R i t) ∨ GetNextPMStep R i t = None"
  shows "¬ ProperMemStep (SM (R t')) (SM (R (Suc t'))) i"
```

And analogously we can deduce that for a run R, a process p_i and for time t if we have a time t' with $t' < t$ and *GetLastPMStep* returns *None* or a value less than t' for R, p_i and t, then there is no ProperMemStep of p_i at t' in R.

```
lemma NoProperMemStepInBetween2: assumes R: "Run R"
  and t': "t' < t"
  and nn: "t' > the(GetLastPMStep R i t) ∨ GetLastPMStep R i t = None"
  shows "¬ ProperMemStep (SM (R t')) (SM (R (Suc t'))) i"
```

As a consequence of Theorem *NoProperMemStepInBetween1*, ProperSteps1, and ProperSteps2 we get that for a run R, a process p_i and for time t if we have a time t' with $t' \geq t$ and *GetNextPMStep* returns *None* or a value greater than t' for R, p_i and t, then the write-oriented view of p_i and the register phase of p_i are equal for t and t'.

```
lemma RegphsInv: assumes R: "Run R"
  and t': "t' ≥ t"
  and nn: "t' < the(GetNextPMStep R i t) ∨ GetNextPMStep R i t = None"
  shows "regphs(SM (R t') i) = regphs(SM (R t) i)"
  "wReg(SM (R t') i) = wReg(SM (R t) i)"
```

And as a consequence of Theorem *NoProperMemStepInBetween2*, ProperSteps1, and ProperSteps2 we get that for a run R, a process p_i and for time t if we have a time t' with $t' \leq t$ and *GetLastPMStep* returns *None* or a value less than t' for R, p_i and t, then the write-oriented view of p_i and the register phase of p_i are equal for t and t'.

```
lemma RegphsInv2: assumes R: "Run R"
  and t': "t' ≤ t"
  and ge: "t' > the(GetLastPMStep R i t)"
  and nn: "¬ GetLastPMStep R i t = None"
  shows "regphs(SM (R t') i) = regphs(SM (R t) i)"
  "wReg(SM (R t') i) = wReg(SM (R t) i)"
```

By the following definition an interval $[begin, end]$ is called a *concurrent access interval* in a run R for a reading process *reader* and a writing process *writer* if there is a time t between *begin* and *end* such that the register phase of *reader* is $\mathbf{rReading}_{writer}$ and the register phase of *writer* is **rWriting** at time t.

```
definition ConcurrentAccessInterval
  :: "(T ⇒ (('a, 'b) conf_scheme)) ⇒ T ⇒ T ⇒ proc ⇒ proc ⇒ bool" where
  "ConcurrentAccessInterval R begin end writer reader ≡ ∃t ≥ begin. t ≤ end
  ∧ regphs (SM (R t) reader) = RReading writer
  ∧ regphs (SM (R t) writer) = RWriting"
```

For a run R, a time t', processes p_i, and p_{regno} the following function `ReadBeginsBefore` returns a set containing all times $t < t'$ such that p_i starts a read operation on $\mathrm{Reg}_{p_{regno}}$ at time t in R.

```
definition
  ReadBeginsBefore
  :: "(T ⇒ (('a, 'b) conf_scheme)) ⇒ T ⇒ proc ⇒ proc ⇒ T set" where
  "ReadBeginsBefore R t' i regno ≡
              {t. t < t' ∧ RegReadBegin (SM (R t)) (SM (R (Suc t))) i regno}"
```

Analogously, for a run R, a time t', a processes p_i the following function `WriteEndsBefore` returns a set containt all times $t < t'$ such that p_i ends a write operation at time t in R.

```
definition
  WriteEndsBefore
  :: "(T ⇒ (('a, 'b) conf_scheme)) ⇒ T ⇒ proc ⇒ T set" where
  "WriteEndsBefore R t' i ≡
              {t. t < t' ∧ RegWriteEnd (SM (R t)) (SM (R (Suc t))) i}"
```

Based on these definitions we define functions `LastReadBegin` (`LastWriteEnd`) that return the most recent access of type read begin (respectively write end) before a given time t' of a process p_i to a register $\mathrm{Reg}_{p_{regno}}$ in a run R.

```
definition LastReadBegin
  :: "(T ⇒ (('a, 'b) conf_scheme)) ⇒ T ⇒ proc ⇒ proc ⇒ T option" where
  "LastReadBegin R t' i regno ≡  if ReadBeginsBefore R t' i regno ≠ {} then
    Some (Max (ReadBeginsBefore R t' i regno))
  else
    None"

definition LastWriteEnd
  :: "(T ⇒ (('a, 'b) conf_scheme)) ⇒ T ⇒ proc ⇒ T option" where
  "LastWriteEnd R t' i ≡  if WriteEndsBefore R t' i ≠ {} then
    Some (Max (WriteEndsBefore R t' i))
  else
    None"
```

In the following, we obtain some simple lemmas for our definitions. Obviously, for a fixed run R, a process p_i and a time t `ReadBeginsBefore` returns always a subset of the set that `ReadBeginsBefore` returns for R, p_i and $t + 1$.

```
lemma RBSubset:
  "ReadBeginsBefore R t' i regno ⊆ ReadBeginsBefore R (Suc t') i regno"
```

Moreover, if `LastReadBegin` returns a value different from `None` for a run R, a time t and process p_i, p_j then `LastReadBegin` also returns a value different from `None` for $R, t+1, p_i$ and p_j.

```
lemma LRBLastingStep:
  assumes R: "Run R"
  and LRB: "LastReadBegin R t i regno ≠ None"
  shows "LastReadBegin R (Suc t) i regno ≠ None"
```

By induction we infer that this does not only hold for $t+1$ but for all $t' \geq t$.

```
lemma LRBLasting:
  assumes R: "Run R"
  and LRB: "LastReadBegin R t i regno ≠ None"
  and t: "t' ≥ t"
  shows "LastReadBegin R t' i regno ≠ None"
```

With the definition of `LastWriteEnd`, we define the function *LastStableVal*, that returns the last stable value of a register before a given time t for a process p_i in a run R (cf. LastStableValue).

```
definition LastStableVal ::
  "(T ⇒ (('a, 'b) conf_scheme))
  ⇒ T
  ⇒ proc
  ⇒ 'content" where
  "LastStableVal R t i ≡ if (LastWriteEnd R t i ≠ None) then
    (ϵc. wReg ((SM (R (Suc (the (LastWriteEnd R t i))))) i) = WriteStable c)
  else
    InitialValue i"
```

If for a run R, for a process p_i and a time t the function `GetLastPMStep` returns another value than for R,p_i and $t+1$, then there must be a ProperMemStep by p_i from t to $t+1$.

```
lemma LastStepChange:
  assumes eq: "GetLastPMStep R i t ≠ GetLastPMStep R i (Suc t)"
  shows "ProperMemStep (SM (R t)) (SM (R (Suc t))) i"
```

If the Shared Memory History changes from time t to $t+1$ in a run R, then some process must have made a ProperMemStep. Hence, we can obtain a fixed process p_i such that p_i executes a ProperMemStep from $R(t)$ to $R(t+1)$.

```
lemma GetMemAction:
  assumes R: "Run R"
  and sm: "SM (R t) ≠ SM (R (Suc t))"
  obtains i where "ProperMemStep (SM (R t)) (SM (R (Suc t))) i"
```

Let R denote a run. If the register phase of a process p_i is $\textbf{rReading}_{p_j}$ in $R(t+1)$ and the phase in $R(t+1)$ differs from the phase in $R(t)$, then p_i must have made a *RdBegin* step from t to $t+1$.

```
lemma NewPhsRReading:
  assumes R: "Run R"
  and phs: "regphs(SM (R t) i) ≠ regphs(SM (R (Suc t)) i)"
           "regphs(SM (R (Suc t)) i) = RReading regno"
  shows "RegReadBegin (SM (R t)) (SM(R (Suc t))) i regno"
```

Analogously, if the register phase of a process p_i is **rWriting** in $R(t+1)$ and the phase in $R(t+1)$ differs from the phase in $R(t)$, then p_i must have made a *WriteBegin* step from t to $t+1$ and we can obtain a value *cnt* such that $\mathit{WriteBegin}(\mathbf{Sh}_{R(t)}, \mathbf{Sh}_{R(t+1)}, p_i, \mathit{cnt})$.

```
lemma NewPhsRWriting:
  assumes R: "Run R"
  and phs: "regphs(SM (R t) i) ≠ regphs(SM (R (Suc t)) i)"
           "regphs(SM (R (Suc t)) i) = RWriting"
  obtains cnt where "RegWriteBegin (SM (R t)) (SM(R (Suc t))) i cnt"
```

If the register phase of a process p_i is **rWriting** in $R(t)$ and the phase in $R(t+1)$ differs from the phase in $R(t)$, then p_i must have made a *WriteEnd* step from t to $t+1$ and the phase in $R(t+1)$ must be **rIdle**.

```
lemma OldPhsRWriting:
  assumes R: "Run R"
  and phs: "regphs(SM (R t) i) ≠ regphs(SM (R (Suc t)) i)"
           "regphs(SM (R t) i) = RWriting"
  shows
  "RegWriteEnd (SM (R t)) (SM(R (Suc t))) i"
  "regphs(SM (R (Suc t)) i) = RIdle"
```

And again, if the register phase of a process p_i is $\mathbf{rReading}_{p_j}$ in $R(t)$ and the phase in $R(t+1)$ differs from the phase in $R(t)$, then p_i must have made a *RdEnd* step from t to $t+1$, the phase in $R(t+1)$ must be **rIdle** and we can obtain a fixed *cnt* such that
$\mathit{RdEnd}(\mathbf{Sh}_{R(t)}, \mathbf{Sh}_{R(t+1)}, p_i, p_j, \mathit{cnt})$.

```
lemma OldPhsRReading:
  assumes R: "Run R"
  and phs: "regphs(SM (R t) i) ≠ regphs(SM (R (Suc t)) i)"
           "regphs(SM (R t) i) = RReading regno"
  obtains cnt
  where
  "RegReadEnd (SM (R t)) (SM(R (Suc t))) i regno cnt"
  "regphs(SM (R (Suc t)) i) = RIdle"
```

Furthermore, if the register phase of a process p_i at time t' is $\mathbf{rReading}_{p_j}$, in a run R we can deduce that `GetLastPMStep` will return the time of the respective *RdBegin* step for R, p_i, t'.

```
lemma phsImplReading1:
  assumes R: "Run R"
  and phs: "regphs (SM (R t') i) = RReading regno"
  shows
  "GetLastPMStep R i t' ≠ None"
  "RegReadBegin
      (SM (R (the(GetLastPMStep R i t'))))
      (SM (R (Suc (the(GetLastPMStep R i t')))))
      i
      regno"
```

If `GetLastPMStep` does not return `None` for a run R, a process p_i and a time t, then the returned value is a fixpoint of `GetNextPMStep` for fixed parameters R and p_i.

```
lemma GetNextLastFix:
  assumes nn: "GetLastPMStep R i t ≠ None"
  shows
  "the (GetNextPMStep R i (the(GetLastPMStep R i t))) = the(GetLastPMStep R i t)"
```

Let R denote a run. If there is a time t' such that the register phase of a process is $\mathbf{rReading}_{p_j}$ in some configuration $R(t')$, then we can obtain a time t with $t < t'$ such that $\mathit{RdBegin}(\mathbf{Sh}_{R(t)}, \mathbf{Sh}_{R(t+1)}, p_i, p_j)$ and `GetNextPMStep` returns either `None` or a value greater or equal than t' for R,p_i and $t + 1$.

```
lemma phsImplReading2:
  assumes R: "Run R"
  and phs: "regphs (SM (R t') i) = RReading regno"
  obtains t
  where
  "t < t'"
  "RegReadBegin (SM (R t)) (SM (R (Suc t))) i regno"
  "the(GetNextPMStep R i (Suc t)) ≥ t' ∨ GetNextPMStep R i (Suc t) = None"
```

If there is a *RdEnd* step by a process p_i at time t in a run R for some register Reg_{p_j}, then `LastReadBegin` returns not `None` for R,t,p_i and p_j.

```
lemma RREImpliesRRB:
  assumes R: "Run R"
  and Res: "RegReadEnd (SM (R t)) (SM (R (Suc t))) i regno cnt"
  shows "LastReadBegin R t i regno ≠ None"
```

If for a run R, a time t and two processes p_i and p_j the function `LastReadBegin` returns a different value than for R, $t + 1$ and p_i, p_j, then there must be a *RdBegin* step in R from t to $t + 1$.

```
lemma LastReadBeginChange:
  assumes eq: "LastReadBegin R t i regno ≠ LastReadBegin R (Suc t) i regno"
  shows "RegReadBegin (SM (R t)) (SM (R (Suc t))) i regno"
```

If in a run R the Shared Memory History changes from t to $t+1$, then we can obtain a process p_i such that p_i makes a ProperMemStep from t to $t+1$.

```
lemma GetProperMemStep:
  assumes R: "Run R"
  and sm: "SM (R t) ≠ SM (R (Suc t))"
  obtains i
  where "ProperMemStep (SM (R t)) (SM (R (Suc t))) i"
```

As for `GetLastPMStep`, we show some basic properties of the function `LastReadBegin`. The following lemma states: if the set `ReadBeginsBefore` is not empty for a run R, a time t and processes p_i, p_j then for R, p_i, p_j and t

- `LastReadBegin` returns not `None`
- `LastReadBegin` returns a time $t' < t$
- there is a *RdBegin* step at the time `LastReadBegin`
- there are no *RdBegin* steps of p_i between the time `LastReadBegin` returns and t

```
lemma LastReadBeginProps:
  assumes ex: "ReadBeginsBefore R t i regno ≠ {}"
  shows
  "LastReadBegin R t i regno ≠ None"
  "the(LastReadBegin R t i regno) < t"
  "RegReadBegin
          (SM (R (the(LastReadBegin R t i regno))))
          (SM (R (Suc (the(LastReadBegin R t i regno)))))
          i
          regno"
  "(∀t''. RegReadBegin (SM (R t'')) (SM (R (Suc t''))) i regno
        ∧ t'' < t ∧ t'' ≠ the(LastReadBegin R t i regno)
      ⟶ t'' < the(LastReadBegin R t i regno))"
```

The following theorem shows an important result for the read-oriented view of a process p_i. Assume the last read operation of p_i for a register Reg_{p_j} in a run R before time t started at $\hat{t} < t$. We show that p_i's read-oriented view on Reg_{p_j} at t is the union of the set of values in p_j's write-oriented view at time $\hat{t}$ and all values for which p_j started a write operation between $\hat{t}$ and t (cf. Theorem 2.3.7.2).

```
theorem ReadResultHelp:
  assumes R: "Run R"
  and nn: "LastReadBegin R t i regno ≠ None"
  shows
  "rRegs (SM(R t) i) regno =
             regValueSet (wReg (SM (R (the(LastReadBegin R t i regno))) regno))
```

```
          ∪ {cnt. ∃t' > the(LastReadBegin R t i regno). t' < t
               ∧ RegWriteBegin (SM (R t')) (SM (R (Suc t'))) regno cnt}"
    thm Max_ge
```

If there is a *WriteEnd* step from t to $t+1$ by a process p_i in a run R, then `LastWriteEnd` returns time t for for R, $t+1$ and i.

```
lemma RWEImpliesLWE:
  assumes RWE: "RegWriteEnd (SM (R t)) (SM (R (Suc t))) i"
  shows "LastWriteEnd R (Suc t) i = Some t"
```

If for a run R, a time t and a processes p_i the function `LastWriteEnd` returns a different value as for R, $t+1$ and p_i, then there must be a *WriteEnd* step in R from t to $t+1$ and `LastWriteEnd` must return time t for R, $t+1$ and p_i.

```
lemma LastWriteEndChange:
  assumes eq: "LastWriteEnd R t i ≠ LastWriteEnd R (Suc t) i"
  shows
  "RegWriteEnd (SM (R t)) (SM (R (Suc t))) i"
  "LastWriteEnd R (Suc t) i = Some t"
```

Analogously to `LastReadBegin`, we show some basic properties of the function `LastWriteEnd`. The following lemma states: if the set `WriteEndsBefore` is not empty for a run R, a time t and processes p_i then for R, p_i and t

- `LastWriteEnd` returns not `None`
- `LastWriteEnd` returns a time $t' < t$
- there is a *WriteEnd* step at the time `LastWriteEnd`
- there are no *WriteEnd* steps of p_i between the time `LastWriteEnd` returns and t

```
lemma LastWriteEndProps: assumes ex: "WriteEndsBefore R t i ≠ {}" shows
  "LastWriteEnd R t i ≠ None"
  "the(LastWriteEnd R t i) < t"
  "RegWriteEnd
          (SM (R (the(LastWriteEnd R t i))))
          (SM (R (Suc (the(LastWriteEnd R t i)))))
          i"
  "(∀t''. RegWriteEnd (SM (R t'')) (SM (R (Suc t''))) i
       ∧ t'' < t ∧ t'' ≠ the(LastWriteEnd R t i)
    ⟶ t'' < the(LastWriteEnd R t i))"
```

Assume the function `LastWriteEnd` returns a value different from `None` for a run R, a time t, and a process p_i and let t' be a time with $t' \geq t$. Then, by the following lemma, `LastWriteEnd` must be different from `None` for R, t' and p_i and, moreover, the returned value must be greater or equal than the value returned for R, t and p_i.

```
lemma LastWriteEndMono:
  assumes lwenn: "LastWriteEnd R t i ≠ None"
  and t: "t ≤ t'"
  shows "LastWriteEnd R t' i ≠ None"
  "the(LastWriteEnd R t i) ≤ the(LastWriteEnd R t' i)"
```

If p_i's write-oriented view is a pair $(\ v, \ v' \ )$ in a run R at time t, then p_i's register phase at time t in R must be **rWriting**.

```
lemma WriteInProgressPhs:
  assumes R: "Run R"
  and wR: "wReg (SM (R t) i) = WriteInProgress cnt cnt'"
  shows
  "regphs (SM (R t) i) = RWriting"
```

With `ReadResultHelp` we characterised the values in the read-oriented view of a process p_i. The following theorem does the same for the write-oriented view of a process. Of course, during a stable phase, the value of p_i's write-oriented view at time t is the *LastStableVal*. During a transient phase this theorem shows that it is a pair (v, v') where v is the *LastStableVal* and v' is a value such that there has been a *WriteBegin* step for v' by p_i before t.

```
theorem wRegContent: assumes R: "Run R" shows
  "wReg (SM (R t) i) = (WriteStable cnt)
        ⟹ cnt = LastStableVal R t i"
  "wReg (SM (R t) i) = (WriteInProgress cnt cnt')
        ⟹ cnt = LastStableVal R t i
          ∧ (∃t' < t. RegWriteBegin (SM (R t')) (SM (R (Suc t'))) i cnt'
          ∧ (the(GetNextPMStep R i (Suc t')) ≥ t
            ∨ GetNextPMStep R i (Suc t') = None))"
```

4.4.3. Equivalence to Lamport's Model

The next theorem shows the first part of our claim that our model implements Lamport's notion of a Reguluar Register. As explained for Theorem 2.3.7.3, this result shows that every value that is read during a *RdEnd* step is a possible candidate for a return of a corresponding read in Lamports model, i.e. it is either the *LastStableVal* for the register or a value that was written during the read operation.

```
theorem ReadResult1:
  assumes R: "Run R"
  and Res: "RegReadEnd (SM (R t)) (SM (R (Suc t))) i regno cnt"
  and c: "ConcurrentAccessInterval R (the(LastReadBegin R t i regno)) t regno i"
  shows
  "(cnt = LastStableVal R (the(LastReadBegin R t i regno)) regno )
  ∨ (∃t'. RegWriteBegin (SM (R t')) (SM (R (Suc t'))) regno cnt ∧
  ((t' > (the(LastReadBegin R t i regno)) ∧ t' < t)
  ∨ (t' < (the(LastReadBegin R t i regno))
    ∧ (GetNextPMStep R regno (Suc t') = None
      ∨ the(GetNextPMStep R regno (Suc t')) > (the(LastReadBegin R t i regno))))))"
```

If a process p_i performs a *RdEnd* step for a register Reg_{p_j} in a run R from t to $t+1$, then for all times t' with $t' \leq t$ and t' after the corresponding *RdBegin* step, the register phase of p_i is $\mathbf{rReading}_{p_j}$.

```
lemma ReadPeriod:
  assumes R: "Run R"
  and Res: "RegReadEnd (SM (R t)) (SM (R (Suc t))) i regno cnt"
  and t: "t' > the(LastReadBegin R t i regno)" "t' ≤ t"
  shows "regphs (SM(R t') i) = RReading regno"
```

If there is no write operation concurrent to a read operation on a register p_j , then the read operation must return the *LastStableVal*(cf. Theorem 2.3.7.4).

```
theorem ReadResult2:
  assumes R: "Run R"
  and Res: "RegReadEnd (SM (R t)) (SM (R (Suc t))) i regno cnt"
  and c: "¬ ConcurrentAccessInterval R (the(LastReadBegin R t i regno)) t regno i"
  shows "cnt = LastStableVal R (the(LastReadBegin R t i regno)) regno"
```

If p_k performs a *WriteBegin* step and the read-oriented view of a process p_i on register Reg_{p_j} is changed, then $p_k = p_j$.

```
lemma RWB_rRegs:
  assumes rwb: "RegWriteBegin c c' k cnt'"
  and PR: "rRegs (c i) j ≠ rRegs (c' i) j"
  shows "k = j"
```

If p_i performs a *RdBegin* step and the read-oriented view of a process p_k is changed, then $p_i = p_k$.

```
lemma RRB_rRegs:
  assumes rwb: "RegReadBegin c c' i j"
  and PR: "rRegs (c k) l ≠ rRegs (c' k) l"
  shows "k = i"
```

If the read-oriented view of a process p_k changes from a Shared Message History **Sh** to **Sh**$'$, then there can be no *RdEnd* and no *WriteEnd* step between **Sh** and **Sh**$'$.

```
lemma RE_rRegs:
  assumes PR: "rRegs (c k) l ≠ rRegs (c' k) l"
  shows
  "∀i j cnt. ¬ RegReadEnd c c' i j cnt"
  "∀i. ¬ RegWriteEnd c c' i"
```

We conclude this theory with the second part of our claim that our model implements Lamport's notion of a reguluar register. As described in Section 2.3.7, if we consider the prefix of a run R where a read operation started by a *RdBegin* step of a process p_i for a register Reg_{p_j} and there is a concurrent write access by p_j where p_j writes a value *cnt* to register Reg_{p_j}, then there is a continuation of R such that p_i reads value *cnt*.

```
theorem ReadNewValuePossible:
  assumes R: "Run R"
  and nn: "LastReadBegin R t i regno ≠ None"
  and phs: "regphs (SM (R t) i) = RReading regno"
  and rwb: "RegWriteBegin (SM (R t')) (SM (R (Suc t'))) regno cnt"
  and t: "(t' > (the(LastReadBegin R t i regno)) ∧ t' < t)
    ∨ (t' < (the(LastReadBegin R t i regno))
      ∧ (GetNextPMStep R regno (Suc t') = None
        ∨ the(GetNextPMStep R regno (Suc t')) > (the(LastReadBegin R t i regno))))"
  obtains c' where "RegReadEnd (SM (R t)) c' i regno cnt"
```

For the second part of our claim, it remains to show that for every prefix of a run R with a start of a read operation there is always a continuation of R where the read operation returns the *LastStableVal* (the 'old' value). This result is represented by our final Theorem `ReadOldValuePossible`.

```
theorem ReadOldValuePossible:
  assumes R: "Run R"
  and nn: "LastReadBegin R t i regno ≠ None"
  and phs: "regphs (SM (R t) i) = RReading regno"
  and cnt: "cnt = LastStableVal R (the(LastReadBegin R t i regno)) regno"
  obtains c' where "RegReadEnd (SM (R t)) c' i regno cnt"

end — SMAlgorithm context

end — theory
```

4.5. Requirements in Isabelle/HOL

The most direct way to show that a predicate $P : \Sigma \rightarrow \mathbf{bool}$ is a property of a distributed algorithm $\mathfrak{A}$ with Isabelle/HOL is to prove a theorem asserting that for every run $R \in \mathrm{Runs}(\mathfrak{A})$ $P(R)$ holds. Figure 4.1 shows how a theorem for Agreement in Isabelle/HOL can be formulated: we assume there is a run `R` and for time t the decision value of the processes `i,j` is not `None` (we use `None` for $\perp$ in Isabelle). It remains to show that the decision value of `i` equals the decision of `j`. Of course, it is also possible to define a predicate for the property first. In Figure 4.2 a predicate `Validity` is defined. Here we take advantage of the fact that Validity is a Configuration invariant and show that in an arbitrary run the expression `∀t. Validity (R t)` is a theorem.
As already mentioned, Termination is a liveness property. Hence, it is not enough to inspect an arbitrary configuration of an arbitrary run R. Instead it is to show that for every arbitrary run

```
theorem Agreement:
  assumes R: "Run R" and
  di: "decision(St (R t) i) ≠ None" and
  dj: "decision(St (R t) j) ≠ None"
  shows "the(decision(St (R t) i)) = the (decision(St (R t) j))"
```

Figure 4.1.: Agreement as a theorem

```
definition
Validity :: "Configuration ⇒ bool" where
"Validity c ≡ ∀i ∈ Procs. ∃k ∈ Procs. St_d (State_S c i) ≠ None ⟶ v_input k =
Decision_v c i"

theorem Validity: assumes R:"R ∈ Runs" shows "∀t. Validity (R t)"
```

Figure 4.2.: Validity as a predicate and Validity as a theorem

```
theorem Termination: assumes R: "Run R" and i: "i ∈ Correct R"
  shows "∃t. decision (St (R t) i) ≠ None"
```

Figure 4.3.: Termination as a theorem

R and for every correct process i there is some t such that in configuration $R(t)$ process i has decided a value (cf. Figure 4.3).

5. Case Studies

In this chapter we will demonstrate the applicability of the introduced model and proof methods on extensive case studies. We examined four well-known algorithms of differing complexity. For each of them, we applied the described methods, specified the formal model and the requirements and verified the correctness of the algorithms formally with Isabelle/HOL. As mentioned before, we decided to place our considerations in the field of Distributed Consensus algorithms. Therefore, all four algorithms deal with the problem of Distributed Consensus, a fundamental concern in distributing computing (cf. Sections 1.4.1 and 2.4.1).

The first algorithm under consideration, the *Rotating Coordinator* Algorithm, is a small example of about 5 lines of pseudo-code, which still features many different parts of our formal model. To make Consensus solvable the Rotating Coordinator algorithm and the algorithm inspected in Section 5.2 utilise *failure detectors*. As explained in Chapter 2, a failure detector can be seen as a module located at every process. This module monitors the crash status of all processes. A system in which the used failure detector modules always monitor the correct crash status of all processes would be almost synchronous. Since we are interested in asynchronous systems, the assumption of having such a *perfect failure* detector induces more synchrony to our system than intended. Literature ([CT96],[CHT96]) shows that imperfect failure detectors, i.e. failure detectors that do not always provide correct information and, hence, are unreliable to a defined degree, are sufficient to solve Consensus in certain settings. The reliability of the used failure detector determines the assumptions on the synchrony of the system. Since the Rotating Coordinator uses a strong failure detector, the induced synchrony leads to a setting where Consensus is solvable by a very simple algorithm. As to be seen, the other algorithms considered in Sections 5.2, 5.3 and 5.4 use much weaker assumptions and hence are more complex examples.

To evaluate how different parts of our model and of our proofs can be reused, we chose very similar algorithms: the *Paxos* algorithm (introduced in [Lam98]) and an algorithm that solves distributed Consensus using failure detector $\Diamond\mathcal{S}$. The latter algorithm and the description of its failure detector are given in [CT96]. Based on the name of the failure detector, in the following we refer to this algorithm as the *$\Diamond\mathcal{S}$-Algorithm.* All three algorithms (Rotating Coordinator, Paxos and the $\Diamond\mathcal{S}$-Algorithm) use message passing mechanisms for communication. For the application of our shared memory model, we choose a fourth algorithm that uses Regular Registers for interprocess communication. This algorithm is embedded in some framework that is called the alpha-abstraction (introduced in [GR07]).

In the following subsections, those four algorithms and the respective models and proofs are introduced and inspected. But at first, let us explain the common setting all algorithms have to face. For all algorithms we assume a fixed number of involved processes (as before, we denote the (finite) set of processes by $\mathbb{P}$ and the number of processes by N). The system is assumed to be asynchronous, i.e., there are no bounds on message delay or process speed. Processes perform atomic steps and communicate by messages (Sections 5.1, 5.2 and 5.3) or by reading and writing to shared memory (Section 5.4). According to our model, atomic process steps and global events generate system runs typically in an interleaving fashion. Messages are not necessarily delivered in the order they are sent. Although there are no synchronous communication rounds, a common characteristic of many distributed algorithms is that there is a notion of asynchronous rounds. This modeling idiom is used to simulate the spirit of synchronized executions, each process keeps and controls a local round number in it its own state. This round number can be attached to messages to give them a logical time stamp. In asynchronous systems, this enables receiving processes to arrange the messages sent by other processes into their intended sequential order [KNR12].

Locally, processes show sequential symmetric behaviour, but possibly hold different initial data. To describe sequential behaviour in terms of our model, we have to map this behaviour to (atomic) actions. Since we use an (unordered) set of actions, we have to add a notion of a program counter to local process states to emulate the sequential character of the algorithm. Fuzzati et al. ([FMN07], [Fuz08]) exemplify the way to achieve this: if the local algorithm is meant to execute actions $\mathcal{A}_0, \ldots, \mathcal{A}_n$ sequentially (i.e. such that each action $\mathcal{A}_i$ is executed before $\mathcal{A}_{i+1}$) we can implement this in our model by introducing *phases* $P_0, \ldots, P_n$. Every process stores the phase it is going to execute next starting with P_0 in its local proces state in a variable *phs*. To implement the sequential behaviour we add a condition ($phs = P_k$) to every action $\mathcal{A}_k$, such that $\mathcal{A}_k$ is only enabled if $phs = P_k$. Additionaly every action $\mathcal{A}_k$ sets the value of *phs* to P_{k+1}. Hence, we can model an action $\mathcal{A}_k$ that reflects the sequential character as:

$$\begin{aligned}\mathcal{A}_k(\mathfrak{C}, \mathfrak{C}', p_i) \triangleq\ & (phs \triangleright \mathbf{S}_{\mathfrak{C}} = P_k)\\ & \wedge \ldots\\ & \wedge (phs \triangleright \mathbf{S}_{\mathfrak{C}'} = P_{k+1})\end{aligned}$$

More details are explained in the following sections.

For each algorithm of our case study we introduce the setting of the algorithm, we explain the algorithm with the help of pseudocode and present the formal model according to our definitions in Chapter 2. Full extensive proofs for correctness and formal models are provided in Isabelle/HOL. Presently, we intend to limit ourselves to special proof issues that are either characteristic for all proofs or special for the respective kind of algorithm and give references to the respective Isabelle theories.

5.1. Rotating Coordinator

The Rotating Coordinator algorithm is introduced in [Tel01]. Francalanza and Hennessy [FH07] use this algorithm to illustrate a process-calculi-based methodology for verifying distributed algorithms. Their approach introduces a *partial failure language* for specification. Proofs are based on a *fault tolerant bisimulation.* The authors claim that, in contrast to their approach, the use of theorem provers for verification of such algorithms ‘tend to obscure the natural structure of the proofs’. With this in mind, we chose to examine this example for accurate analysis. Although it is only a short algorithm, it exhibits many of the typical aspects that are integral to tolerant distributed computing:

- It deals with an asynchronous setting with crash failures.
- It uses a point-to-point message passing communication infrastructure.
- Safety and liveness properties have to be regarded.
- The algorithm works round-based and uses a coordinator to manage rounds.
- The algorithm uses a failure detector for making Consensus solvable

The *Rotating Coordinator* solves Consensus by using the *strong* failure detector $\mathcal{S}$ introduced in [CT96]. The assumption for this failure detector is that the information provided by the failure detector may be wrong but will at least satisfy two conditions [CT96]:

1. **Weak accuracy**: Some correct process is never suspected.
2. **Strong completeness**: Eventually every process that crashes will be permanently suspected to have crashed.

In a Crash-Failure model (cp. Section 2.3.8) a process is said to be *correct* in a run R if it does not crash in R. Hence, by the weak accuracy property, in every run R there must be at least one process that all processes can trust to be alive and which does actually not crash in R. Since this process does not crash and is trusted by all other processes to be alive, we call such a process a *trusted-immortal* process. Usually, an algorithm that uses a failure detector with strong completeness stops waiting for a message if the sender of the awaited message is suspected. Hence, the strong completeness property prevents processes from waiting eternally for a message from a crashed process because every crashed process is eventually suspected [Tel01]. Thus, the failure detector $\mathcal{S}$ guarantees that there is at least one trusted-immortal process and that no process waits eternally for messages from crashed processes.

For the message infrastructure, we assume that messages cannot be lost in this setting and a message that has been sent must eventually be delivered.

In a setting with N processes, this algorithm solves Consensus up to $N - 1$ crash failures. Francalanza and Hennessy [FH07] identify two error conditions that the algorithm needs to resolve: 1. processes waiting for messages that will never arrive due to crashed senders and 2. broadcast messages that are only received by a subset of the participants because the sender crashed before the sending of the message had been finished for all receivers. It is easy to see that the failure detector helps to overcome the first error condition. As we will see in the following subsections, the second error condition can be resolved by repeating the broadcast.

5.1.1. Informal Introduction to the Algorithm

Listing 5.1 shows a pseudo code that captures the local behaviour of each of the N participants. We assume that every process initially gets an input value from a set of possible input values $\mathbb{I}$. This value is stored in a variable `Input`

Initially, a participant p_i sets a local variable `x_i` to its input value `Input`. Then the algorithm performs N rounds. A variable `r` is used to represent a corresponding round number in a *for* loop that runs from 1 to N. For simplicity, it is assumed that processes are numbered consecutively from p_1 to p_N such that each round r can be assigned to process p_r so that p_r is the coordinator of round r. In each round, p_i first checks whether it is the coordinator of the current round r and, if this is the case, then p_i broadcasts its current value x_i. Note that in contrast to the defined reliable broadcast in Section 2.3.6, *broadcast* in this context means just sending a (point-to-point) message to every process. If the failure detector gives notice that the coordinator of p_i's current round r is alive, p_i waits for the message of the coordinator. After p_i received this message, p_i sets x_i to the value that was broadcasted by the coordinator. After N rounds, p_i decides the value that it has stored in x_i.

Listing 5.1: The Rotating Coordinator Algorithm for participant p_i [FH07]

```
1 x_i := Input
2 for r:= 1 to N do {
3       if r = i then broadcast(x_i);
4       if alive(p_r) then x_i := input_from_broadcast;
5 }
6 output(x_i);
```

Intuitively, this algorithm solves Consensus by utilising the guarantees of the failure detector. It is easy to see that the Agreement property can be violated, if the processes always skip the receipt of messages by assuming the coordinator not to be alive. In this case, every process p_i would decide its own input value Input. Consequently, Agreement would be violated for the cases where input values vary between the processes. Since trusted-immortal processes are not suspected to have crashed, the function alive always returns true in rounds of trusted-immortal processes. Skipping messages of trusted-immortal processes is therefore not possible. Let p_{ti} be a trusted-immortal process (the failure detector property for $\mathcal{S}$ assures that there is at least one). Since skipping the message of p_{ti} is not possible in round $ti + 1$, every process must have adopted the value x_{ti} that p_{ti} broadcasted to x_i in round ti. Therefore, in rounds $ti + 1 \dots N$, every broadcasted value must be x_{ti} and therefore x_{ti} is the only value that can be adopted in rounds after round ti. Each process that reaches a decision (line 6) must have processed these rounds (and, at least, must have processed the message of p_{ti}). Hence, the value a process decides in line 6 must be x_{ti}. Thus, Agreement holds under the assumption that there is at least one trusted-immortal process.

Processes use only input values for values in x_i and x_i values are only adopted from messages. Also for the values in messages only input values are used. Hence, Validity can not be violated.

There can only be a finite number of rounds to process before the decision is made. Hence, if a correct process p_c neither blocks nor crashes (note that correct process do not crash), it must reach a decision. The only point where a blocking of p_c is possible, would be in line 4, where p_c waits for a message of the coordinator that is supposed to be alive. Assume for contradiction r_b is the smallest roundnumber in which a correct process p_c blocks forever in line 4. Due to the Strong completeness property of the failure detector, the sender of the message p_c waits for must be correct, otherwise alive would eventually have returned False. Let p_b be the delayed sender of this message. Since r_b is the smallest roundnumber in which a correct process is blocked, p_b cannot block in a roundnumber smaller than r_b. Furthermore, as deduced before, p_b is a correct process and hence can not crash. Thus p_b must eventually reach round p_b and send a message m with its value x_i to all processes including p_c. Since every sent message is eventually received,

p_c must eventually receive m and, in contradiction to the assumption, no longer blocks in line 4 of round r_b. Hence, Termination holds.

5.1.2. Formal Model

For our formal model we first need to define the datastructures for configurations, Localviews and messages. For this simple algorithm we use two entries in our configurations:

- $\mathbf{S}$: the array of process states
- $\mathbf{Q}$: the Message History

As usual, we denote the array of process states of a configuration $\mathfrak{C}$ by $\mathbf{S}_{\mathfrak{C}}$ and the Message History of $\mathfrak{C}$ by $\mathbf{Q}_{\mathfrak{C}}$. The module for the Message History (without message loss) is used as introduced in Chapter 2.3.5. The state of a process p_i is represented by a tuple $(\ r,\ P,\ x_i,\ c,\ d\ )$ where

- $r \in \mathbb{N}$ is the current round number of p_i.
- $P \in \{\ \mathrm{P1},\ \mathrm{P2}\ \}$ is the current phase of p_i.
- $x_i \in \mathbb{I}$ is the current value of p_i (corresponds to x_i in the pseudo code).
- $c \in \mathbf{bool}$ is the (boolean) crash status of p_i (see Section 2.3.8).
- $d \in \mathbb{I} \cup \{\ \perp\ \}$ is the decision value of p_i. This value is undefined ($\perp$) when the algorithm starts and is set to the decided value when p_i decides.

Again, we use the introduced means for notation to denote the entries of a process state $\mathbf{S}_{\mathfrak{C}}(p_i)$ of a process p_i in a configuration $\mathfrak{C}$:

- the round number of p_i is denoted by $rnd \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the phase of p_i is denoted by $phs \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the value of p_i is denoted by $x \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the crash status of p_i is denoted by $crashed \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the decision value of p_i is denoted by $dcsn \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$

In a Localview lv a process can access its process state $\mathbf{S}_{lv}$ and the respective entries $rnd \triangleright \mathbf{S}_{lv}, phs \triangleright \mathbf{S}_{lv}, x \triangleright \mathbf{S}_{lv}, crashed \triangleright \mathbf{S}_{lv}, dcsn \triangleright \mathbf{S}_{lv}$. Furthermore, the Localview comprises a set $\mathbf{Out}_{lv}$ for the outbox of the process, the inbox $\mathbf{In}_{lv}$ of p_i, (the set of received messages of p_i) and the identity $\mathbf{ID}_{lv}$ of p_i. While we expect the values of the inbox and the identity to be read-only values, we model the outbox as an emptyset where p_i puts in the messages p_i is willing to send (as already explained and used in Sections 2.3.2 and 2.3.5).

A message m is a tuple $(\ s,\ r,\ x\ )$ where

- $s \in \mathbb{P}$ is the sender of m

- $r \in \mathbb{P}$ is the receiver of m
- $x \in \mathbb{I}$ is the content of m (which is for this algorithm the value x of the respective coordinator).

The sender of a message m is denoted by snd_m, the receiver by rcv_m, and the content by cnt_m. Initially, every process p_i starts with round number 1 in phase P1 with its input value $\text{v}_{\text{inp}}(p_i)$ (which corresponds to x_iin the pseudocode), alive (hence not crashed) and undecided. Furthermore, the Message History contains no messages and hence returns zero for every pair of messages and message status values. We define the initial process state $\mathbf{S}_0(p_i)$ for each process p_i, the initial Message History $\mathbf{Q}_0$ and finally the set of initial states Init_{rc} as depicted in figure 5.1.

$$\mathbf{S}_0(p_i) \triangleq (\ 1,\ \text{P1},\ \text{v}_{\text{inp}}(p_i),\ \text{False},\ \perp\ ) \qquad\qquad \mathbf{Q}_0(m,t) \triangleq 0$$

$$\text{Init}_{\text{rc}} \triangleq \left\{ \begin{pmatrix} \mathbf{S}_0 \\ \mathbf{Q}_0 \end{pmatrix} \right\}$$

Figure 5.1.: Initial definitions for Rotating Coordinator Algorithm

Now we are ready to define the actions for the algorithm. Initialization is already done by the definitions in figure 5.1. Hence, there remain three steps, that matter for a process p_i:

- In the first step p_i sends messages to all processes if p_i is the coordinator of its current round $rnd \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.
- In the second step either p_i trusts the coordinator of the current round and processes the respective message or p_i suspects the coordinator to be crashed, skips waiting for the message and transits to the next round.
- Finally, the decision must be taken after N rounds.

Figure 5.2 depicts how we use non-lifted actions to model these steps. The first step is represented by the DnA $\boldsymbol{MsgGen}^{\downarrow}$, which consists of the enabled condition $MsgGen^{\downarrow}_{en}$ and the transition function $MsgGen^{\downarrow}_{\Delta}$. As already explained, for serialization of the steps we use two phases P1 and P2. We use P1 to indicate that we are at the beginning of the for-loop of the pseudocode and hence $MsgGen^{\downarrow}$ is therefore only enabled if the phase of the executing process is P1. Furthermore, the process must not be crashed and the current round of the process is less than or equal to N. $MsgGen^{\downarrow}_{\Delta}$ describes the transition to the next phase: all values in the process state are unchanged but the phase is set to P2. Furthermore, for the case that the executing process is the coordinator of the current round, a message for every process containing the current x value of the coordinator is put into the outbox. Otherwise, the outbox is the emptyset. The Inbox and identity are (always) unchanged by process-actions.

$$\begin{aligned}
\mathit{MsgGen}^{\downarrow}_{en}(lv) &\triangleq (\mathit{rnd} \triangleright \mathbf{S}_{lv} \leq N) \wedge (\mathit{phs} \triangleright \mathbf{S}_{lv} = \mathrm{P1}) \wedge (\neg \mathit{crashed} \triangleright \mathbf{S}_{lv}) \\
\mathit{Msgs}_{id,x}(r) &\triangleq \begin{cases} \{\ (\ id,\ p_j,\ x\ )\ \mid\ p_j \in \mathbb{P}\ \} & \text{, if } \mathrm{PID}(id) = r \\ \emptyset & \text{, else} \end{cases} \\
\mathit{MsgGen}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \mathrm{P2},\ x \triangleright \mathbf{S}_{lv},\ \mathit{crashed} \triangleright \mathbf{S}_{lv},\ \mathit{dcsn} \triangleright \mathbf{S}_{lv}\ ), \\ \mathit{Msgs}_{\mathbf{ID}_{lv}, x \triangleright \mathbf{S}_{lv}}, \\ \mathbf{In}_{lv}, \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{MsgGen}^{\downarrow} &\triangleq (\ \mathit{MsgGen}^{\downarrow}_{en},\ \mathit{MsgGen}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

$$\begin{aligned}
\mathit{MsgRcvTrust}^{\downarrow}(lv, lv') \triangleq\ & \exists m \in \mathbf{In}_{lv}\ .\ \mathrm{PID}(\mathrm{snd}_m) = \mathit{rnd} \triangleright \mathbf{S}_{lv} \wedge (\mathit{phs} \triangleright \mathbf{S}_{lv} = \mathrm{P2}) \\
& \wedge (\neg \mathit{crashed} \triangleright \mathbf{S}_{lv}) \\
& \wedge \mathbf{S}_{lv'} = (\mathit{rnd} \triangleright \mathbf{S}_{lv} + 1, \mathrm{P1}, \mathrm{cnt}_m, \mathit{crashed} \triangleright \mathbf{S}_{lv}, \mathit{dcsn} \triangleright \mathbf{S}_{lv}) \\
& \wedge \mathbf{Out}_{lv'} = \emptyset
\end{aligned}$$

$$\begin{aligned}
\mathit{MsgRcvSuspect}^{\downarrow}_{en}(lv) &\triangleq (\mathit{rnd} \triangleright \mathbf{S}_{lv} \neq \mathrm{PID}(\mathbf{ID}_{lv})) \\
&\quad \wedge (\mathit{phs} \triangleright \mathbf{S}_{lv} = \mathrm{P2}) \\
&\quad \wedge (\neg \mathit{crashed} \triangleright \mathbf{S}_{lv}) \\
&\quad \wedge (p_{\mathit{rnd} \triangleright \mathbf{S}_{lv}} \notin \mathrm{TI}) \\
\mathit{MsgRcvSuspect}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ \mathit{rnd} \triangleright \mathbf{S}_{lv} + 1,\ \mathrm{P1},\ x \triangleright \mathbf{S}_{lv},\ \mathit{crashed} \triangleright \mathbf{S}_{lv},\ \mathit{dcsn} \triangleright \mathbf{S}_{lv}\ ), \\ \mathbf{Out}_{lv}, \\ \mathbf{In}_{lv}, \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{MsgRcvSuspect}^{\downarrow} &\triangleq (\ \mathit{MsgRcvSuspect}^{\downarrow}_{en},\ \mathit{MsgRcvSuspect}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

$$\begin{aligned}
\mathit{Finish}^{\downarrow}_{en} &\triangleq \mathit{rnd} \triangleright \mathbf{S}_{lv} > N \\
&\quad \wedge (\mathit{phs} \triangleright \mathbf{S}_{lv} = \mathrm{P1}) \\
&\quad \wedge (\neg \mathit{crashed} \triangleright \mathbf{S}_{lv}) \\
&\quad \wedge (\mathit{dcsn} \triangleright \mathbf{S}_{lv} \neq \bot) \\
\mathit{Finish}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \mathrm{P1},\ x \triangleright \mathbf{S}_{lv},\ \mathit{crashed} \triangleright \mathbf{S}_{lv},\ x \triangleright \mathbf{S}_{lv}\ ), \\ \mathbf{Out}_{lv}, \\ \mathbf{In}_{lv}, \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{Finish}^{\downarrow} &\triangleq (\ \mathit{Finish}^{\downarrow}_{en},\ \mathit{Finish}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

Figure 5.2.: Actions and Events for Rotating Coordinator Algorithm (Part I)

The non-lifted action ***MsgRcvTrust***$^{\downarrow}$ chooses a message to process. Although by the definition of the algorithm it is clear that there can only be one possible message, in a general setting we have to assume that there might be more than one message arriving at process p_i such that p_i must choose one message to process out of a set of messages. Therefore, without further restrictions, this message must be chosen nondeterministically. Hence, this action is not modeled as a DnA but as a simple non-lifted action. The action is enabled if there is a message for the current round of the process, the process is not crashed, and the phase is P2. If the action is executed, the content of the message is adopted to the state of the process and the phase is set back to P1.

We model the weak accuracy property by a nonempty set TI that contains the trusted-immortal processes. Note that TI is fixed for a run R and hence is not part of a configuration but can be modeled as a constant.

The DnA ***MsgRcvSuspect***$^{\downarrow}$ models the case where a process suspects the coordinator of its current round. Therefore, to be enabled, the coordinator of the current round must not be in set TI (note that the coordinator of round r is p_r). Of course, this action can also be executed only if the process is not crashed and the phase is P2. Moreover, a process is not allowed to suspect itself and therefore a process p_i suspects only coordinators of rounds j with $j \neq i$. This action does nothing more than incrementing the round number by one and setting the phase to P1.

Finally, after N rounds have been completed, a process is able to decide by executing the DnA ***Finish***$^{\downarrow}$. *Finish*$^{\downarrow}$ is enabled if a round number greater than N is reached, the phase is P1, the process has not yet decided, and is not crashed. To decide, a process simply copies the value from the x value to the decision value and terminates. Note that after all processes either decided or have crashed and all possible message movements have been done, the system reaches a deadlock (there are no enabled actions or events any more).

To model crashes we provide an DnA ***Crash***$^{\downarrow}$ (see figure 5.3), which is always enabled if the process has not crashed already and is not a trusted-immortal process. In the case of crash-failure the indicating boolean value $\mathit{crashed} \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$ of the process is set to True and everything else is unchanged. Note that the state of being crashed is a sink, i.e. a process that has crashed in a run R by the definition of the Crash-Failure model will never recover in R. Finally, we define two events to manage message movements from one message status to the next (cf. section 2.3.5). ***MsgSend*** shifts an **outgoing** message to the messages with status **transit** and ***MsgDeliver*** advances a message from **transit** to status **received**. These two events use the introduced subactions Transmit and Receive (see section 2.3.5). Note that from a model theoretic point of view it is not obvious that Transmit and Receive are events because they require a process (for Transmit the sender process and for Receive the receiver process) to be alive. Hence, they could be regarded as process-actions of this process. But defining these actions as process-actions would require to make **transit** messages accessible for processes, which does not conform to our concept of Localviews. Therefore, we decided to model *MsgSend* and *MsgDeliver* as events. Figure 5.4 shows how we convert configurations to Localviews and vice versa. The function $\mathrm{C2LV}_{\mathrm{rc}}$ generates a Localview of a configuration $\mathfrak{C}$ for a process p_i by returning a vector of the

- state of a process: $\mathbf{S}_{\mathfrak{C}}(p_i)$

$$
\begin{aligned}
\mathit{Crash}^{\downarrow}_{en} &\triangleq (\neg \mathit{crashed} \triangleright \mathbf{S}_{lv}) \\
&\quad \wedge (\mathbf{ID}_{lv} \notin \mathrm{TI}) \\
\mathit{Crash}^{\downarrow}_{\Delta} &\triangleq \begin{pmatrix} (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \mathit{phs} \triangleright \mathbf{S}_{lv},\ x \triangleright \mathbf{S}_{lv},\ \mathrm{True},\ \mathit{dcsn} \triangleright \mathbf{S}_{lv}\ ), \\ \mathbf{Out}_{lv}, \\ \mathbf{In}_{lv}, \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{Crash}^{\downarrow} &\triangleq (\ \mathit{Crash}^{\downarrow}_{en},\ \mathit{Crash}^{\downarrow}_{\Delta}\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{MsgSend}(\mathfrak{C}, \mathfrak{C}') &\triangleq \exists m\ .\ \mathrm{Transmit}(\mathbf{Q}_{\mathfrak{C}}, \mathbf{Q}_{\mathfrak{C}'}, m) \\
&\quad \wedge (\neg \mathit{crashed} \triangleright \mathbf{S}_{\mathfrak{C}}(\mathrm{snd}_m)) \\
&\quad \wedge (\mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{MsgDeliver}(\mathfrak{C}, \mathfrak{C}') &\triangleq \exists m\ .\ \mathrm{Receive}(\mathbf{Q}_{\mathfrak{C}}, \mathbf{Q}_{\mathfrak{C}'}, m) \\
&\quad \wedge (\neg \mathit{crashed} \triangleright \mathbf{S}_{\mathfrak{C}}(\mathrm{rcv}_m)) \\
&\quad \wedge (\mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}})
\end{aligned}
$$

Figure 5.3.: Actions and events for Rotating Coordinator Algorithm (Part II)

$$
\begin{aligned}
\mathrm{C2LV}_{\,\mathrm{rc}}(\mathfrak{C}, p_i) &\triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}(p_i) \\ \emptyset \\ \{\ m \in \mathrm{recmsgs}_{\,\mathbf{Q}_{\mathfrak{C}}}\ \mid\ \mathrm{rcv}_m = p_i\ \} \\ p_i \end{pmatrix} \\
\mathrm{LV2C}_{\,\mathrm{rc}}(lv, p_i, \mathfrak{C}) &\triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{lv}] \\ \mathrm{sndupd}_{\mathbf{Q}_{\mathfrak{C}}, \mathbf{Out}_{lv}} \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
\Phi_{\mathrm{rc}} &\triangleq \left\{\ \mathrm{DLift}(\mathit{MsgGen}^{\downarrow}),\ \mathrm{Lift}(\mathit{MsgRcvTrust}^{\downarrow}),\ \mathrm{DLift}(\mathit{MsgRcvSuspect}^{\downarrow})\ \right\} \\
&\quad \cup \left\{\ \mathrm{DLift}(\mathit{Finish}^{\downarrow}),\ \mathrm{DLift}(\mathit{Crash}^{\downarrow})\ \right\} \\
\Psi_{\mathrm{rc}} &\triangleq \{\ \mathit{MsgSend},\ \mathit{MsgDeliver}\ \} \\
\mathfrak{A}_{\mathrm{rc}} &\triangleq (\ \mathrm{Init}_{\mathrm{rc}},\ \Phi_{\mathrm{rc}},\ \Psi_{\mathrm{rc}},\ \emptyset,\ \mathrm{C2LV}_{\,\mathrm{rc}},\ \mathrm{LV2C}_{\,\mathrm{rc}}\ )
\end{aligned}
$$

Figure 5.4.: Formal Rotating Coordinator Algorithm

- an empty outbox: $\emptyset$
- the received messages of p_i in $\mathfrak{C}$: $\{\ m \in \mathrm{recmsgs}_{\,\mathbf{Q}_{\mathfrak{C}}}\ \mid\ \mathrm{rcv}_m = p_i\ \}$

- the identity of p_i: p_i

LV2C_{rc} embeds a Localview lv back in to a given configuration $\mathfrak{C}$ by

- setting the state of process p_i to the process state $\mathbf{S}_{lv}$ of the Localview
- updating the **outgoing** messages by adding the messages from p_i's outbox (note that sndupd is the function defined in Section 2.3.5)

Finally we lift the defined actions and define Φ_{rc} as the set of all lifted actions. Ψ_{rc} is defined as the set of events *MsgSend* and *MsgDeliver*. We conclude with the tuple
$(\ \text{Init}_{\text{rc}},\ \Phi_{\text{rc}},\ \Psi_{\text{rc}},\ \emptyset,\ \text{C2LV}_{\text{rc}},\ \text{LV2C}_{\text{rc}}\ )$ that defines the formal Distributed Algorithm $\mathfrak{A}_{\text{rc}}$ (see figure 5.4).

5.1.3. Proof Issues

The algorithm aims to solve Consensus. Hence, in order to prove correctness, we have to verify the properties defined in Section 2.4.1. Figure 5.5 shows the properties in terms of our formal model. Proofs for correctness are given in $\hookrightarrow$ Isabelle[1]. The idea for the proof of Termination is, to show that in every run R there is a time t_d such that there is a deadlock in configuration $R(t_d)$ and every correct process has a round number greater than N ($\hookrightarrow$ Isabelle[2]). As described in Section 5.1.1, the proof requires to show that no trusted-immortal process p_{ti} blocks before p_{ti} reaches round ti ($\hookrightarrow$ Isabelle[3]). Verification of these lemmas are not as easy as it may seem. First, different subgoals have to be proven, as for example, that every run R is finite ($\hookrightarrow$ Isabelle[4]) and different lemmas concerning the enabledness of actions. Having established that every correct process p_c has a round number greater than N in time t_d, it is easy to show that p_c must have decided on a value. This implies Termination.

$$\begin{aligned}
\text{Validity}_{\text{rc}}(R) &\triangleq \forall t \in \mathbb{T}\,.\,\forall p_i \in \mathbb{P}\,.\, \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \Rightarrow \big(\exists p_k \in \mathbb{P}\,.\, \text{v}_{\text{inp}}(p_k) = \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_i)\big)\\
\text{Agreement}_{\text{rc}}(R) &\triangleq \forall t \in \mathbb{T}\,.\,\forall p_i, p_j \in \mathbb{P}\,.\, \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \wedge \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_j) \neq \bot\\
&\qquad \Rightarrow \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_i) = \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_j)\\
\text{Termination}_{\text{rc}}(R) &\triangleq \forall p_i \in \text{Correct}(R)\,.\,\exists t \in \mathbb{T}\,.\, \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot\\
\text{Irrevocability}_{\text{rc}}(R) &\triangleq \forall t \in \mathbb{T}\,.\,\forall p_i \in \mathbb{P}\,.\, \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot\\
&\qquad \Rightarrow \big(\forall t' \in \mathbb{T}\,.\, \big(t' \geq t \Rightarrow \mathit{dcsn} \triangleright \mathbf{S}_{R(t')}(p_i) = \mathit{dcsn} \triangleright \mathbf{S}_{R(t)}(p_i)\big)\big)
\end{aligned}$$

Figure 5.5.: Formal Properties of Distributed Consensus for the Rotating Coordinator Algorithm

Let p_{ti} be a trusted-immortal process. The proof of Agreement relies mainly on four arguments:

[1]RotatingCoordVerification.thy
[2]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `CorrectProcs_reach_MaxPCHelp`
[3]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `NonBlockingTI`
[4]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `OnlyFiniteRuns`

- Processes only decide if they have reached a round number greater than N ($\hookrightarrow$ Isabelle[5]).
- All processes have equal x values in rounds higher than ti ($\hookrightarrow$ Isabelle[6]).
- If a process decides then it will decide for its x value ($\hookrightarrow$ Isabelle[7]).
- $ti \leq N$.

Naturally, the most effort will be spent in proving the second argument.

Finally, Validity and Irrevocability are shown by a simple application of the methods described in Section 3.2.

In respect to our Isabelle proofs we do not comply the argument of Francalanza and Hennessy that the natural structure of the proofs would be somehow obscured by the application of a theorem prover. On the contrary, the use of Isabelle and the corresponding proof style Isar enables us to transcribe the proofs in a very similar fashion to the paper proof given in [Tel01]. They differ only in regards to the level of detail.

5.2. $\Diamond\mathcal{S}$-Algorithm

While the previous section presented an algorithm that solves Distributed Consensus by using failure detector $\mathcal{S}$, this section considers an example where the weaker failure detector $\Diamond\mathcal{S}$ is used. As indicated by '$\diamond$', this failure detector eventually shows the properties of $\mathcal{S}$ (hence, it is called the *Eventually Strong* failure detector). The failure detector used here satisfies the following two conditions [CT96]:

1. **Eventual weak accuracy**: There is a time, after which some correct process is never suspected by any correct proccess.
2. **Strong completeness**: Eventually, every process that crashes is permanently suspected to be crashed.

The algorithm under inspection is introduced in [CT96]. Since assumptions for the failure detector are weaker, the algorithm is much more complex. Moreover, some additional assumptions are needed to solve Consensus:

- More than the half of all involved processes must be correct.
- The communication infrastructure must provide a Quasi-reliable point-to-point mechanism and a reliable broadcast mechanism (see Sections 2.3.5 and 2.3.6).

Note that while variants of the algorithm with weaker assumptions for the communication infrastructure are possible, the first assumption is mandatory for an algorithm that uses an Eventually Strong (and even an Eventually Perfect) failure detector to solve Consensus (a proof is provided in [CT96] and [Tel01]). [CT96] showed that a system that uses a failure detector with weak

[5]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `DecidedAfterN`
[6]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `uniformRndsAfterTI2`
[7]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `Consistency`

completeness (and weak accuracy) is equivalent to a system that uses a failure detector with strong completeness (and weak accuracy). In the companion paper [CHT96] it is shown that the failure detector with weak completeness (and weak accuracy) is the weakest failure detector that solves Consensus. Therefore, there can be no algorithm that solves Distributed Consensus in an asynchronous setting with crash failures that uses a failure detector that is effectively weaker than the one used here. Compared to the Rotating Coordinator algorithm, the $\Diamond\mathcal{S}$-algorithm exhibits another feature that is important for most applications. While the Rotating Coordinator algorithm might violate the Agreement property if the failure detector fails to provide weak accuracy, this algorithm guarantees the safety properties of Consensus even in the case where the failure detector violates the assumed guarantees and even if more than the half of all processes fail. The eventual weak accuracy property makes it undecidable to predetermine the time that is needed to solve Consensus and this, in contrast to the Rotating Coordinator algorithm, leads to an unknown number of rounds that are required in a run to reach Termination. Hence for this algorithm the maximum number of rounds that are needed to solve Distributed Consensus is finite in every run but can not be determined any further. Like the Rotating Coordinator algorithm, this algorithm is round based and uses the *rotating coordinator* paradigm [Fuz08]: for each round, one process plays the role of the coordinator of r. Based on collected messages from [other] processes, the coordinator of a round proposes a value for decision. Since there might be more than N rounds, the role of the coordinator of round r can no longer be assigned to p_r. Therefore we use the function

$$\mathsf{Crd(r)} \triangleq ((\mathsf{r}{-}1) \mod N) + 1$$

to determine the coordinator of round r. For more detailed information on this algorithm the reader is refered to [CT96] and [Fuz08].

5.2.1. Informal Introduction to the Algorithm

The pseudo code (listing 5.2) describes the local behaviour of a process p_i that executes the $\Diamond\mathcal{S}$ algorithm. We adopt the listing from [NF03] instead of the original from [CT96] because it is closer to our formalization.

The algorithm uses a directive QPP_SEND(snd,(round,phase),c)**TO** rcv for quasi-reliable point-to-point communication. This directive is supposed to send a message from the sender snd to the receiver rcv. The pair (round,phase) is used as a logical time stamp that allows the receiver to arrange messages in order. A message can be uniquely identified by the tuple (snd, rcv, $(round, phase)$) because every sender passes each round only once and in each round each phase is only visited once or never. c is used for the content of the message. We assume that received messages are stored in a set received_QPP and that we can access all received messages from round number r and phase P as set $\mathsf{received\_QPP.P^r}$.

To send a message d with the reliable broadcast mechanism, a process snd uses the directive RBC_BROADCAST(snd,d). After a broadcast d from a process *snd* has been delivered, the receiving process is noticed by RBC_DELIVER(snd,d).

Listing 5.2: $\Diamond\mathcal{S}$- Algorithm for participant p_i [Fuz08]

```
r             := 0
belief.value  := Input
belief.stamp  := 0
decision      := ⊥

while (decision = ⊥)
{ W   r := r + 1
  P1  QPP_SEND(p_i,(r,P1),belief) TO Crd(r)

  P2  if    (p_i = Crd(r)) then {
          await |received_QPP.P1^r| ≥ ⌈(N+1)/2⌉
          for 1 ≤ j ≤ N do
              QPP_SEND(p_i,(r,P2),best(received_QPP.P1^r)) TO p_j
      }
  P3  await ( received_QPP.P2^r ≠ ∅ or (¬ alive(Crd(r))) )
      if (received_QPP.P2^r = {(Crd(r),(r,P2),(v,s))}) then {
          QPP_SEND(p_i,(r,P3),ACK) TO Crd(r)
          belief.value := v
          belief.stamp := r
      } else {
          QPP_SEND(i,(r,P3),NACK) TO Crd(r)
      }
  P4  if (p_i = Crd(r)) then {
          await |received_QPP.P3^r| ≥ ⌈(N+1)/2⌉
          if |ack(received_QPP.P3^r)| ≥ ⌈(N+1)/2⌉ then RBC_BROADCAST(p_i, belief)
      }
}



when (RBC_DELIVER (p_j, d) {
  if (decision = ⊥) then {
    decision := d
    output(decision.value)
  }
}
```

Locally, each process makes use of variables belief, r, v, and j. belief and decision are records with entries value and stamp, such that a value from the set of input values and a round number can be stored in belief.value and belief.stamp (respectively decision.value and decision.stamp). For the decision value it is also possible to be undefined ($\perp$) to represent an undecided process. The belief of a process p_i represents p_i's current belief, i.e., a value (belief.value) the process recently adopted as a candidate for the decision value and a stamp belief.stamp denoting the round in which p_i adopted that value.

Initially, the round number r and belief.stamp are set to zero, belief.value is set to the input value of the process, and the decision value is set to $\perp$. Then, in a while-loop, processes cycle through four phases $\mathrm{P1}, \mathrm{P2}, \mathrm{P3}, \mathrm{P4}$ (note that for our formalization we added a fifth phase W that was originally not introduced by Chandra and Toueg but derives from the formalization in

[FMN07]). Before entering phase P1, the round number r is incremented by one. In phase P1 of a round r every process p_i sends its current belief to the coordinator of r. In phase P2, if p_i is the coordinator of r then p_i collects the sent beliefs of at least $\lceil \frac{N+1}{2} \rceil$ processes and determines the value to propose as a decision value by a function best. Among all received belief values, best selects the value with the most recent stamp, i.e., the greatest stamp value associated with the highest round number. To wait for at least $\lceil \frac{N+1}{2} \rceil$ P1 messages of round r the statement **await** is used. **await** b is assumed to suspend the local execution of the program until the (boolean) condition b is satisfied. The best value is then sent to all processes. In phase P3, p_i waits until either it gets the P2 message for the current round from the coordinator or the function alive indicates that the coordinator has crashed. In case p_i has received the respective P2 message, p_i sends an acknowledgement ACK to the coordinator, adopts the proposed value for belief.value, and sets belief.stamp to the current round. If p_i suspects the coordinator to have crashed, p_i sends a NACK message to the coordinator to indicate that p_i does not acknowledge the value of the coordinator. Phase P4 is only executed if p_i is the coordinator of the current round. In this case p_i waits for at least $\lceil \frac{N+1}{2} \rceil$ P3 messages. If p_i gets more than $\lceil \frac{N+1}{2} \rceil$ acknowledgements then p_i will broadcast its current belief (the value p_i proposed in phase P2) for decision. Then p_i continues at the beginning of the while-loop. Concurrently to the while-loop, processes wait for broadcasts and hence the pseudo code after line 28 must be interpreted as a concurrent sub-routine. If a process p_i receives a broadcast (indicated by RBC_DELIVER) then p_i decides for the broadcasted value v by applying v for the local decision value and outputting v.

As a higher level description of the algorithm, Chandra and Toueg [CT96] explain that the algorithm passes through three asynchronous epochs. In the first epoch, all input values of the processes are possible candidates for the decision. During the second epoch, one input value v gets *locked*: A value is locked for a round r if more than the half of all processes acknowledged the value sent by the coordinator of r and from then on no other decision value will be possible. In the third episode, processes decide for v. Coordinators use a maximal time-stamp strategy and always wait for a majority of acknowledgements. Therefore a locked value v will always dominate the other values[Fuz08], which is the main argument for the Agreement property. Also for this algorithm the main argument for Validity is that no values are invented. All values in messages and in process states originate from the input values of processes. Finally, Termination depends on the assumption that at least a majority of all processes have not crashed, because this implies that a coordinator does not wait forever for a majority of messages. Hence, coordinators do not block while they wait for messages. On the other hand, the other processes are able to skip waiting for messages of crashed coordinators by suspecting them. Therefore, every possible round number can be reached by correct processes. If no decision has been taken before, eventually a trusted-immortal process p_c, that is not suspected, will be coordinator of a round r. In this round r no (correct) process suspects p_c. Thus, p_c collects more than $N/2$ acknowledgements and broadcasts its value such that all correct processes must decide eventually by the assumption of the Reliable Broadcast mechanism.

5.2.2. Formal Model

As for the previous algorithm, we begin with the introduction of the required datastructures for configurations, Localviews and messages. Concepts for the datastructures are mainly influenced by the definitions of [FMN07], therefore, for more details of the single components, we refer the reader to Fuzzati et al. A configuration for our $\Diamond\mathcal{S}$-algorithm has five entries:

- **S**: the array of process states
- **TI**: the set of trusted-immortal processes
- **B**: the set of broadcast messages
- **Q**: the Message History
- **Crashed**: the set of crashed processes.

In contrast to the Rotating Coordinator the set of trusted-immortal processes for this algorithm is not a constant function but modeled as a part of the configuration. The concept here is that a process can become trusted-immortal at any point in time and an additional fairness assumption *OmegaProperty* assures that eventually there is some process in the set of trusted-immortal processes. This failure detector model originates from the concepts introduced in [NF03] and actually models failure detector Ω, which is equivalent to $\Diamond\mathcal{S}$ (a proof is given in [CHT96] and [NF03]).

Again, we denote the array of process states of a configuration $\mathfrak{C}$ by $\mathbf{S}_{\mathfrak{C}}$, the set of trusted-immortal processes by $\mathbf{TI}_{\mathfrak{C}}$, the set of broadcast messages by $\mathbf{B}_{\mathfrak{C}}$, and the set of crashed processes by $\mathbf{Crashed}_{\mathfrak{C}}$. We use the same module for the Message History as for the Rotating Coordinator model but add an additional fairness property QPPReliability (see Section 2.3.5). For broadcasts we use the modeling techniques introduced in Section 2.3.6. As described in Section 2.3.8 we keep track of crash failures by a set of processes. The state of a process p_i is represented by a tuple $(\ (\ r,\ P\ ),\ (\ v,\ s\ ),\ d\ )$, where

- $(\ r,\ P\ )$ is the program counter of p_i consisting of the round number r of p_i and the phase P p_i is in.
- $(\ v,\ s\ )$ is the belief of p_i consisting of a value $v \in \mathbb{I}$ and a stamp $s \in \mathbb{N}$ (see section 5.2.1).
- d is the decision value of p_i. d must be either $\perp$ (undecided) or a pair $(\ v_d,\ s_d\ )$ where v_d is the value p_i decided and s_d is the stamp of the round where the decision was taken.

Again, we use the introduced means for notation to denote the entries for a process state $\mathbf{S}_{\mathfrak{C}}(p_i)$ of a process p_i in a configuration $\mathfrak{C}$:

- the program counter of a process is denoted by $pc \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the phase of p_i is denoted by $phs \triangleright pc \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the round number of p_i is denoted by $rnd \triangleright pc \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the belief of p_i is denoted by $bel \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the value of the belief of p_i is denoted by $val \triangleright bel \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$

- the stamp of the belief of p_i is denoted by $\mathit{stamp} \triangleright \mathit{bel} \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the decision of p_i is denoted by $\mathit{dc} \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the value of the decision of p_i is denoted by $\mathit{val} \triangleright \mathit{dc} \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the stamp of the decision of p_i is denoted by $\mathit{stamp} \triangleright \mathit{dc} \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$

In a Localview lv a process can access its process state $\mathbf{S}_{lv}$ and the respective entries $\mathit{pc} \triangleright \mathbf{S}_{lv}$, $\mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}$, $\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}$ $\mathit{bel} \triangleright \mathbf{S}_{lv}$, $\mathit{val} \triangleright \mathit{bel} \triangleright \mathbf{S}_{lv}$, $\mathit{stamp} \triangleright \mathit{bel} \triangleright \mathbf{S}_{lv}$, $\mathit{dc} \triangleright \mathbf{S}_{lv}$, $\mathit{val} \triangleright \mathit{dc} \triangleright \mathbf{S}_{lv}$, and $\mathit{stamp} \triangleright \mathit{dc} \triangleright \mathbf{S}_{lv}$. As for the Rotating Coordinator algorithm, the Localview comprises the process state, a set $\mathbf{Out}_{lv}$ for the outbox of the process, the inbox $\mathbf{In}_{lv}$ of p_i, (the set of received messages of p_i), and the identity $\mathbf{ID}_{lv}$ of p_i. Again, we expect the values of the inbox and the identity to be read-only values and we model the outbox as an emptyset where p_i puts in the messages p_i is willing to send (as already explained and used in Sections 2.3.2 and 2.3.5). For the $\Diamond\mathcal{S}$-algorithm we add an entry $\mathbf{B}_{lv}$ for the Broadcast History, a boolean value $\mathbf{isCrashed}_{lv}$ for the crash status of p_i, and an entry $\mathbf{Trusted}_{lv}$ for the set of trusted-immortal processes.

Messages are represented by tuples $(\ s,\ r,\ \mathit{rnd},\ \mathit{phs},\ c\ )$ where

- $s \in \mathbb{P}$ is the sender of m
- $r \in \mathbb{P}$ is the receiver of m
- $\mathit{rnd} \in \mathbb{N}$ is the round number the message is associated with
- $\mathit{phs} \in \{\ \mathrm{P1},\ \mathrm{P2},\ \mathrm{P3},\ \mathrm{P4}\ \}$ is the phase the message is associated with
- c is an arbitrary content of the message

The sender of a message m is denoted by snd_m, the receiver by rcv_m, the round by rnd_m, the phase by P_m, and the content of a message is denoted by cnt_m.

Initially, every process p_i starts with round number 0 and phase W, with the belief $(\ \mathrm{v_{inp}},\ 0\ )$ and p_i is initially undecided. Furthermore, the Message History contains no messages and hence returns zero for every message/message-status pair. Therefore, we define the initial process state $\mathbf{S}_0(p_i)$ for each process p_i, the initial Message History $\mathbf{Q}_0$, and finally the set of initial states $\mathrm{Init_{ct}}$ as depicted in Figure 5.6. The emptyset entries in $\mathrm{Init_{ct}}$ represent (in this order) the emptyset of trusted-immortal processes, the emptyset of broadcast messages, and the emptyset of crashed processes at initialization.

As annotated in the pseudo code, we divide this algorithm into five phases $\mathrm{W}, \mathrm{P1}, \mathrm{P2}, \mathrm{P3}$, and P4. For modeling the actions, we transfered the transition rules for the algorithm given in [FMN07] to our model. Only the addition of one action for the failure detector ($\mathit{ImmoTrust}^{\downarrow}$) and a few minor changes were necessary. Hence, the reader is also refered to [FMN07] for additional information on the single actions.

$$\mathbf{S}_0(p_i) \triangleq (\ (\ 0,\ \mathrm{W}\ ),\ (\ \mathrm{v}_{\mathrm{inp}},\ 0\ ),\ \bot\ )$$
$$\mathbf{Q}_0(m,t) \triangleq 0$$

$$\mathrm{Init}_{\mathrm{ct}} \triangleq \left\{ \begin{pmatrix} \mathbf{S}_0 \\ \emptyset \\ \emptyset \\ \mathbf{Q}_0 \\ \emptyset \end{pmatrix} \right\}$$

Figure 5.6.: Initial definitions for $\Diamond\mathcal{S}$-Algorithm

The phases reflect the main steps of the algorithm:

- If the process p_i is undecided in the first step (phase W), p_i increments the round number and proceeds to phase P1.
- As a second step, p_i sends a message with its belief to the coordinator of the round. If p_i is the coordinator of the round, p_i proceeds to phase P2 else p_i proceeds to phase P3.
- If p_i is in phase P2 (and therefore is the coordinator of its current round), p_i waits for P1 messages from the processes. If p_i has received messages from more than a half of all processes, p_i determines the *best* belief (i.e. the belief with the most recent stamp) and sends it as a proposal to all processes. Then p_i proceeds to phase P3
- If p_i suspects the coordinator of the current round in phase P3, p_i sends a NACK message to the coordinator and proceeds to phase P1. Otherwise, p_i waits for the P2 message of the coordinator, adopts the sent belief value for its own belief, sends a ACK message and proceeds either to phase P4 (if p_i is the coordinator of the round) or to W otherwise.
- If p_i is in phase P4 (and therefore is the coordinator of its current round), p_i collects P3 messages from the processes. If p_i has received acknowledgements from more than a half of all processes, p_i broadcasts its belief as decision value. p_i proceeds to phase W.

In our model, the first step is represented by the DnA ***While***$^{\downarrow}$ (see Figure 5.7). This action is enabled, if the executing process is not crashed, is in phase W and has not yet decided. The transition increments the round number and sets the phase to P1, everything else in the Localview is unchanged.

Then, in phase P1 DnA ***Phs1***$^{\downarrow}$ is enabled if the process does not crash in between. *Phs1*$^{\downarrow}$ generates a P1 message to the coordinator with the current belief of the process and puts it into the outbox. A function $\mathrm{nxP}(r, P, p_i)$ determines the next phase dependent on the coordinator of

r and the phase P of process p_i. In case of $P = \text{P1}$ this can be either P2 (if p_i is the coordinator of r) or P3 (otherwise).

$$\begin{aligned}
\mathit{While}^{\downarrow}_{en}(lv) &\triangleq \neg\mathbf{isCrashed}_{lv} \\
&\quad \wedge \mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} = \text{W} \\
&\quad \wedge \mathit{dc} \triangleright \mathbf{S}_{lv} = \perp \\
\mathit{While}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} + 1,\ \text{P1}\ ),\ \mathit{bel} \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv} \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathbf{isCrashed}_{lv} \\ \mathbf{Trusted}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{While}^{\downarrow} &\triangleq (\ \mathit{While}^{\downarrow}_{en},\ \mathit{While}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

$$\begin{aligned}
\text{nxP}(r, P, p_i) &\triangleq \begin{cases} \text{P2} & \text{, if } P = \text{P1} \wedge \text{Crd}(r) = p_i \\ \text{P3} & \text{, if } P = \text{P1} \wedge \text{Crd}(r) \neq p_i \\ \text{P4} & \text{, if } P = \text{P3} \wedge \text{Crd}(r) = p_i \\ \text{W} & \text{, if } P = \text{P3} \wedge \text{Crd}(r) \neq p_i \end{cases} \\
\mathit{Phs1}^{\downarrow}_{en}(lv) &\triangleq \neg\mathbf{isCrashed}_{lv} \\
&\quad \wedge \mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} = \text{P1} \\
\mathit{Phs1}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \text{nxP}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \text{P1},\ \mathbf{ID}_{lv})\ ),\ \mathit{bel} \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv} \\ \{\ (\ \mathbf{ID}_{lv},\ \text{Crd}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}),\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \text{P1},\ \mathit{bel} \triangleright \mathbf{S}_{lv}\ )\ \} \\ \mathbf{B}_{lv} \\ \mathbf{isCrashed}_{lv} \\ \mathbf{Trusted}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{Phs1}^{\downarrow} &\triangleq (\ \mathit{Phs1}^{\downarrow}_{en},\ \mathit{Phs1}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

Figure 5.7.: Actions $\mathit{While}^{\downarrow}$ and $\mathit{Phs1}^{\downarrow}$

In phase P2 the DnA $\boldsymbol{\mathit{Phs2}}^{\downarrow}$ is enabled if the process is not crashed and has already received P1 messages from more than a half of all processes for the current round (see Figure 5.8). Among those messages the executing process selects the belief with the most recent stamp by using the following functions:

- highest(r, M) returns the highest (most recent) stamp of all P1 messages from set M with round number r (note that the stamp is the 2nd entry of the content of a P1 message).
- winner(r, M) returns the process id j of the process p_j that sent the message with the

highest stamp. Since in general this process is not unique, we chose the one with the minimal process id.

- best(r, M) returns the belief that corresponds to the winner message.

This belief is then sent to all processes in a P2 message and the process transits to phase P3.

$$\begin{aligned}
\mathit{Phs2}^{\downarrow}_{en}(lv) &\triangleq \neg\mathbf{isCrashed}_{lv} \\
&\quad \wedge \mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} = \mathrm{P2} \\
&\quad \wedge |\{\ m \in \mathbf{In}_{lv} \ \mid\ \exists c\ .\ \exists p_j \in \mathbb{P}\ .\ m = (\ p_j,\ \mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P1},\ c\ )\ \}| > \frac{N}{2} \\
\mathrm{highest}(r, M) &\triangleq \mathrm{Max}\{\ s \ \mid\ \exists m \in M\ .\ \mathrm{rnd}_m = r \wedge \mathrm{P}_m = \mathrm{P1} \wedge s = 2^{\mathrm{nd}}\,(\mathrm{cnt}_m)\ \} \\
\mathrm{winner}(r, M) &\triangleq \mathit{Min}\{\ j \ \mid\ \exists m \in M\ .\ \mathrm{rnd}_m = r \wedge \mathrm{P}_m = \mathrm{P1} \wedge \mathrm{snd}_m = p_j \wedge 2^{\mathrm{nd}}\,(\mathrm{cnt}_m) = \mathrm{highest}(r, M)\ \} \\
\mathrm{best}(r, M) &\triangleq \epsilon b.\ \exists m \in M\ .\ \mathrm{rnd}_m = r \wedge \mathrm{P}_m = \mathrm{P1} \wedge \mathrm{snd}_m = p_{\mathrm{winner}(r,M)} \wedge \mathrm{cnt}_m = b \\
\mathit{Phs2}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix}
(\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P3}\ ),\ \mathit{bel} \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ ) \\
\mathbf{In}_{lv} \\
\{\ (\ \mathbf{ID}_{lv},\ p_j,\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P2},\ \mathrm{best}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathbf{In}_{lv})\ )\ \mid\ p_j \in \mathbb{P}\ \} \\
\mathbf{B}_{lv} \\
\mathbf{isCrashed}_{lv} \\
\mathbf{Trusted}_{lv} \\
\mathbf{ID}_{lv}
\end{pmatrix} \\
\mathit{Phs2}^{\downarrow} &\triangleq (\ \mathit{Phs2}^{\downarrow}_{en},\ \mathit{Phs2}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

Figure 5.8.: Action *Phs2*$^{\downarrow}$

Dependent on the messages a process p_i has received, in phase P3 two actions can be enabled for p_i: ***Phs3Trust***$^{\downarrow}$ or ***Phs3Suspect***$^{\downarrow}$ (see Figure 5.9). If p_i has received a P2 message from the coordinator of the current round, then p_i trusts the coordinator and *Phs3Trust*$^{\downarrow}$ is enabled. Otherwise, if the coordinator is not a trusted-immortal process and p_i itself is not the coordinator of the current round, p_i may suspect the coordinator to have crashed and execute *Phs3Suspect*$^{\downarrow}$. As for the Rotating Coordinator algorithm, the selection of the message is modeled non-deterministically (see Section 5.1.2) and hence *Phs3Trust*$^{\downarrow}$ is not modeled as a DnA but as a simple non-lifted action. In contrast to *Phs3Trust*$^{\downarrow}$, action *Phs3Suspect*$^{\downarrow}$ sends a NACK to the coordinator. If p_i is the coordinator of the current round, the phase is set to P4 (by using again function nxP). Otherwise, the phase is set back to W to start a new round.

Also for a process in phase P4 two of the *PhsX* actions can be enabled: ***Phs4Success***$^{\downarrow}$ and ***Phs4Fail***$^{\downarrow}$ (see figure 5.10). Both actions are modeled as DnAs. *Phs4Success*$^{\downarrow}$ is executed by a coordinator p_i if p_i received P3 acknowledgements from more than the half of all processes. In this case p_i broadcasts its belief to make the other processes decide. Otherwise, if p_i received P3 messages from more than the half of all processes but not enough acknowledgements, p_i executes *Phs4Fail*$^{\downarrow}$. In both cases p_i transits to phase W.

$$\begin{aligned}
\mathit{Phs3Trust}^{\downarrow}(lv, lv') \triangleq\; & \neg\mathbf{isCrashed}_{lv} \\
& \wedge \mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} = \mathrm{P3} \\
& \wedge \mathbf{In}_{lv'} = \mathbf{In}_{lv} \\
& \wedge \mathbf{B}_{lv'} = \mathbf{B}_{lv} \\
& \wedge \mathbf{isCrashed}_{lv'} = \mathbf{isCrashed}_{lv} \\
& \wedge \mathbf{Trusted}_{lv'} = \mathbf{Trusted}_{lv} \\
& \wedge \exists m \in \mathbf{In}_{lv} \;.\; (\mathrm{rnd}_m = \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} \wedge \mathrm{P}_m = \mathrm{P2} \\
& \wedge \mathbf{Out}_{lv'} = \{\ (\ \mathbf{ID}_{lv},\ \mathrm{Crd}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}),\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ \mathrm{ACK}\ )\ \} \\
& \wedge \mathbf{S}_{lv'} = \\
& \quad (\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{nxP}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ \mathbf{ID}_{lv})\ ),\ (\ 1^{\mathrm{st}}\,(\mathrm{cnt}_m),\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}\ )\ ))
\end{aligned}$$

$$\begin{aligned}
\mathit{Phs3Suspect}^{\downarrow}_{en}(lv) \triangleq\; & \neg\mathbf{isCrashed}_{lv} \\
& \wedge \mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} = \mathrm{W} \\
& \wedge \mathrm{Crd}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}) \notin \mathbf{Trusted}_{lv} \\
& \wedge \mathbf{ID}_{lv} \neq \mathrm{Crd}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}) \\
& \wedge \forall m \in \mathbf{In}_{lv} \;.\; \mathrm{rnd}_m = \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} \Rightarrow \mathrm{P}_m \neq \mathrm{P2}
\end{aligned}$$

$$\mathit{Phs3Suspect}^{\downarrow}_{\Delta}(lv) \triangleq \begin{pmatrix} (\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{W}\ ),\ \mathit{bel} \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv} \\ \{\ (\ \mathbf{ID}_{lv},\ \mathrm{Crd}(\mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv}),\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ \mathrm{NACK}\ )\ \} \\ \mathbf{B}_{lv} \\ \mathbf{isCrashed}_{lv} \\ \mathbf{Trusted}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix}$$

$$\mathit{Phs3Suspect}^{\downarrow} \triangleq (\ \mathit{Phs3Suspect}^{\downarrow}_{en},\ \mathit{Phs3Suspect}^{\downarrow}_{\Delta}\ )$$

Figure 5.9.: Actions *Phs3Trust*$^{\downarrow}$ and *Phs3Suspect*$^{\downarrow}$

For the concurrent task of processing broadcasts, we define an action ***RBCDeliver***$^{\downarrow}$ (see Figure 5.11). As for point-to-point messages, the selection of the broadcast is carried out non-deterministically and therefore this action is not modeled as a DnA but as a standard non-lifted action. A process p_i processes a broadcast independendly from the phase and therefore the only conditions for this action to be enabled are that p_i is not crashed and that there is a broadcast for p_i available that has not been processed yet. Processing a broadcast means adopting the value for decision (if p_i has not yet decided a value).

As for the Rotating Coordinator, we model crashes by providing an DnA ***Crash***$^{\downarrow}$, which is always enabled if the process has not yet already crashed and is not a trusted-immortal process. In the case of crash-failure the variable $\mathbf{isCrashed}_{lv'}$ for the process is set to True.

$$
\begin{aligned}
\mathit{Phs4Fail}^{\downarrow}_{en}(lv) \triangleq{}& \neg\mathbf{isCrashed}_{lv}\\
&\wedge \mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} = \mathrm{P4}\\
&\wedge |\{\ m \in \mathbf{In}_{lv}\ \mid\ \exists c\ .\ \exists p_j \in \mathbb{P}\ .\ m = (\ p_j,\ \mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ c\ )\ \}| > \frac{N}{2}\\
&\wedge |\{\ m \in \mathbf{In}_{lv}\ \mid\ \exists p_j \in \mathbb{P}\ .\ m = (\ p_j,\ \mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ \mathrm{ACK}\ )\ \}| \le \frac{N}{2}\\
\mathit{Phs4Fail}^{\downarrow}_{\Delta}(lv) \triangleq{}& \begin{pmatrix} (\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{W}\ ),\ \mathit{bel} \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ )\\ \mathbf{In}_{lv}\\ \mathbf{Out}_{lv}\\ \mathbf{B}_{lv}\\ \mathbf{isCrashed}_{lv}\\ \mathbf{Trusted}_{lv}\\ \mathbf{ID}_{lv} \end{pmatrix}\\
\mathit{Phs4Fail}^{\downarrow} \triangleq{}& (\ \mathit{Phs4Fail}^{\downarrow}_{en},\ \mathit{Phs4Fail}^{\downarrow}_{\Delta}\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Phs4Success}^{\downarrow}_{en}(lv) \triangleq{}& \neg\mathbf{isCrashed}_{lv}\\
&\wedge \mathit{phs} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv} = \mathrm{P4}\\
&\wedge |\{\ m \in \mathbf{In}_{lv}\ \mid\ \exists p_j \in \mathbb{P}\ .\ m = (\ p_j,\ \mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ \mathrm{ACK}\ )\ \}| > \frac{N}{2}\\
\mathit{Phs4Success}^{\downarrow}_{\Delta}(lv) \triangleq{}& \begin{pmatrix} (\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ \mathrm{W}\ ),\ \mathit{bel} \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ )\\ \mathbf{In}_{lv}\\ \mathbf{Out}_{lv}\\ \mathbf{B}_{lv} \cup \{\ (\ \mathit{rnd} \triangleright \mathit{pc} \triangleright \mathbf{S}_{lv},\ 1^{\mathrm{st}}\,(\mathit{bel} \triangleright \mathbf{S}_{lv}),\ \mathbb{P}\ )\ \}\\ \mathbf{isCrashed}_{lv}\\ \mathbf{Trusted}_{lv}\\ \mathbf{ID}_{lv} \end{pmatrix}\\
\mathit{Phs4Success}^{\downarrow} \triangleq{}& (\ \mathit{Phs4Success}^{\downarrow}_{en},\ \mathit{Phs4Success}^{\downarrow}_{\Delta}\ )
\end{aligned}
$$

Figure 5.10.: Actions $\mathit{Phs4Success}^{\downarrow}$ and $\mathit{Phs4Fail}^{\downarrow}$

An DnA ***ImmoTrust***$^{\downarrow}$ is used to let processes become trusted-immortal. As described in [NF03] a process can become trusted-immortal, if it is not crashed and has not yet become already trusted-immortal. The execution of *ImmoTrust*$^{\downarrow}$ adds the process to the set of trusted-immortal processes.

As for the Rotating Coordinator, we define two actions ***MsgSend*** and ***MsgDeliver*** to manage message movements from one message status to the next (see Section 2.3.5). *MsgSend* shifts an **outgoing** message to the messages with status **transit** and *MsgDeliver* advances a message from **transit** to status **received** (see Figure 5.12). These two events use the introduced subactions Transmit and Receive (see Section 2.3.5). As for the Rotating Coordinator algorithm we decided to model *MsgSend* and *MsgDeliver* as events.

Figure 5.13 shows the definitions for the functions $\mathrm{C2LV_{ct}}$ and $\mathrm{LV2C_{ct}}$ that are used to convert configurations to Localviews and vice versa. The function $\mathrm{C2LV_{ct}}$ generates a Localview out of a configuration $\mathfrak{C}$ for a process p_i by returning a vector of the

- state of a process: $\mathbf{S}_{\mathfrak{C}}(p_i)$
- the inbox of p_i, i.e. the received messages of p_i in $\mathfrak{C}$: $\{\ m \in \mathrm{recmsgs}_{\,\mathbf{Q}_{\mathfrak{C}}} \ \mid\ \mathrm{rcv}_m = p_i\ \}$
- an emptyset for the outbox
- the Broadcast History
- the value **isCrashed** that is the boolean value of the assertion ($p_i \in \mathbf{Crashed}_{\mathfrak{C}}$)
- the set of trusted-immortal processes
- the identity of p_i: p_i

$\mathrm{LV2C_{ct}}$ embeds a Localview lv back in to a given configuration $\mathfrak{C}$ by

- setting the state of process p_i to the process state $\mathbf{S}_{lv}$ of the Localview
- adding p_i to the set of trusted-immortal processes if p_i was added locally to the respective set
- adopting the local Broadcast History to the global
- updating the **outgoing** messages by adding the messages from p_i's outbox (note that sndupd is the function defined in Section 2.3.5)
- adding p_i to the crashed processes if p_i's variable $\mathbf{isCrashed}_{lv}$ is set to True

To define the required fairness assumptions for the $\Diamond\mathcal{S}$-algorithm, we first formally define the set of **correct** processes of a run R (see Figure 5.14). According to our definition, a process p_i is correct in R if and only if p_i is never in set **Crashed** in R.

Based on this definition we define the fairness assumptions ***RBCValidity***, ***RBCAgreement***, ***OmegaProperty***, and ***CorrectPBound***. *RBCValidity* and *RBCAgreement* are the formal assertions for the guarantees explained in Section 2.3.6:

$$\begin{aligned}
\mathrm{sdc}(v,r,d) &\triangleq \begin{cases} d & \text{, if } d \neq \bot \\ (v,r) & \text{, else} \end{cases} \\
\mathit{RBCDeliver}^{\downarrow}_{en}(lv,lv') &\triangleq \neg\mathbf{isCrashed}_{lv} \\
&\wedge \mathbf{Out}_{lv'} = \mathbf{Out}_{lv} \\
&\wedge \mathbf{In}_{lv'} = \mathbf{In}_{lv} \\
&\wedge \mathbf{isCrashed}_{lv'} = \mathbf{isCrashed}_{lv} \\
&\wedge \mathbf{Trusted}_{lv'} = \mathbf{Trusted}_{lv} \\
&\wedge \exists b \in \mathbf{B}_{lv} \;.\; (\mathbf{ID}_{lv} \in 3^{\mathrm{rd}}\,(b) \\
&\wedge \mathbf{B}_{lv'} = (\mathbf{B}_{lv} \setminus b) \cup \left\{\, (1^{\mathrm{st}}\,(bc),\; 2^{\mathrm{nd}}\,(bc),\; 3^{\mathrm{rd}}\,(bc) \setminus \mathbf{ID}_{lv}) \,\right\} \\
&\wedge \mathbf{S}_{lv'} = (\; \mathit{pc} \triangleright \mathbf{S}_{lv},\; \mathit{bel} \triangleright \mathbf{S}_{lv},\; \mathrm{sdc}(1^{\mathrm{st}}\,(b),\; 2^{\mathrm{nd}}\,(b),\; \mathit{dc} \triangleright \mathbf{S}_{lv})\;))
\end{aligned}$$

$$\begin{aligned}
\mathit{Crash}^{\downarrow}_{en}(lv) &\triangleq \neg\mathbf{isCrashed}_{lv} \\
&\wedge \mathbf{ID}_{lv} \in \mathbf{Trusted}_{lv} \\
\mathit{Crash}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} \mathbf{S}_{lv} \\ \mathbf{In}_{lv} \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathrm{True} \\ \mathbf{Trusted}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{Crash}^{\downarrow} &\triangleq (\; \mathit{Crash}^{\downarrow}_{en},\; \mathit{Crash}^{\downarrow}_{\Delta}\;)
\end{aligned}$$

$$\begin{aligned}
\mathit{ImmoTrust}^{\downarrow}_{en}(lv) &\triangleq \neg\mathbf{isCrashed}_{lv} \\
&\wedge \mathbf{ID}_{lv} \notin \mathbf{Trusted}_{lv} \\
\mathit{ImmoTrust}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} \mathbf{S}_{lv} \\ \mathbf{In}_{lv} \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathbf{isCrashed}_{lv} \\ \mathbf{Trusted}_{lv} \cup \{\, \mathbf{ID}_{lv} \,\} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{ImmoTrust}^{\downarrow} &\triangleq (\; \mathit{ImmoTrust}^{\downarrow}_{en},\; \mathit{ImmoTrust}^{\downarrow}_{\Delta}\;)
\end{aligned}$$

Figure 5.11.: Actions $\mathit{RBCDeliver}^{\downarrow}$, $\mathit{Crash}^{\downarrow}$ and $\mathit{ImmoTrust}^{\downarrow}$

$$
\begin{aligned}
\mathit{QPPSnd}(\mathfrak{C}, \mathfrak{C}') \triangleq\ & \exists m \,.\, \mathrm{snd}_m \notin \mathbf{Crashed}_{\mathfrak{C}} \\
& \land \mathrm{Transmit}(\mathbf{Q}_{\mathfrak{C}}, \mathbf{Q}_{\mathfrak{C}'}, m) \\
& \land \mathbf{Crashed}_{\mathfrak{C}'} = \mathbf{Crashed}_{\mathfrak{C}} \\
& \land \mathbf{TI}_{\mathfrak{C}'} = \mathbf{TI}_{\mathfrak{C}} \\
& \land \mathbf{B}_{\mathfrak{C}'} = \mathbf{B}_{\mathfrak{C}} \\
& \land \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{QPPDeliver}(\mathfrak{C}, \mathfrak{C}') \triangleq\ & \exists m \,.\, \mathrm{rcv}_m \notin \mathbf{Crashed}_{\mathfrak{C}} \\
& \land \mathrm{Receive}(\mathbf{Q}_{\mathfrak{C}}, \mathbf{Q}_{\mathfrak{C}'}, m) \\
& \land \mathbf{Crashed}_{\mathfrak{C}'} = \mathbf{Crashed}_{\mathfrak{C}} \\
& \land \mathbf{TI}_{\mathfrak{C}'} = \mathbf{TI}_{\mathfrak{C}} \\
& \land \mathbf{B}_{\mathfrak{C}'} = \mathbf{B}_{\mathfrak{C}} \\
& \land \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}}
\end{aligned}
$$

Figure 5.12.: Events *QPPSnd* and *QPPDeliver*

$$
\mathrm{C2LV}_{\mathrm{ct}}(\mathfrak{C}, p_i) \triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}(p_i) \\ \{\, m \in \mathrm{recmsgs} \mid \mathrm{rcv}_m = p_i \,\} \\ \emptyset \\ \mathbf{B}_{\mathfrak{C}} \\ (p_i \in \mathbf{Crashed}_{\mathfrak{C}}) \\ \mathbf{TI} \\ p_i \end{pmatrix}
$$

$$
\mathrm{LV2C}_{\mathrm{ct}}(lv, p_i, \mathfrak{C}) \triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{lv}] \\ (\mathbf{TI}_{\mathfrak{C}} \cup \{\, p_j \mid p_j = p_i \land p_i \in \mathbf{Trusted}_{lv} \,\}) \\ \mathbf{B}_{lv} \\ \mathrm{sndupd}_{\mathbf{Q}_{\mathfrak{C}}, \mathbf{Out}_{lv}} \\ (\mathbf{Crashed}_{\mathfrak{C}} \cup \{\, p_j \mid p_j = p_i \land \mathbf{isCrashed}_{lv} \,\})\, \mathbf{B}_{lv} \end{pmatrix}
$$

Figure 5.13.: Functions $\mathrm{C2LV}_{\mathrm{ct}}$ and $\mathrm{LV2C}_{\mathrm{ct}}$

RBCValidity If a correct process p_i broadcasts a message m, then p_i eventually delivers m.

RBCAgreement If a broadcast message m is delivered by some correct process, then m is eventually delivered by every process.

The guarantees ‘No Duplication’ and ‘No Creation’ are implicitly given by the definition of our broadcast model and therefore need no explicit declaration. Our fairness assumption for *ReliableBroadcast* is therefore given by the conjunction of *RBCValidity* and *RBCAgreement*.

The *OmegaProperty* is the formal declaration of our assumption on the failure detector: There is a process p_k such that there is a time $t \in \mathbb{T}$ where p_k is in the set of trusted-immortal processes. Finally, *CorrectPBound* requires that at least more than the half of all processes are correct.

The definition of the set of predicates $\mathcal{F}_{\text{ad}}$ that must hold for Admissible Runs contains the three aforementioned predicates *ReliableBroadcast*, *OmegaProperty*, *CorrectPBound*, and additionally the predicate for QPPReliability that has been defined in Section 2.3.5.

$$\text{Correct}(R) \triangleq \{\, p_i \in \mathbb{P} \;\mid\; \forall t \in \mathbb{T} \,.\, p_i \notin \mathbf{Crashed}_{R(t)} \,\}$$

$$\begin{aligned}
\mathit{RBCValidity}(R) \triangleq{}& \forall p_i \in \text{Correct}(R) \,.\, \forall t \in \mathbb{T} \,.\, \forall b \in \mathbf{B}_{R(t)} \,.\\
& (\text{DLift}(\mathit{Phs4Success}^{\downarrow}))(R(t-1), R(t), p_i) \wedge t > 0 \wedge b \notin \mathbf{B}_{R(t-1)} \;\Rightarrow\\
& \exists u \in \mathbb{T} \,.\, u > t \wedge \exists b' \in \mathbf{B}_{R(u-1)} \,.\, \exists b'' \in \mathbf{B}_{R(u)} \,.\\
& \quad \wedge\, 1^{\text{st}}(b') = 1^{\text{st}}(b) \wedge 1^{\text{st}}(b'') = 1^{\text{st}}(b)\\
& \quad \wedge\, 2^{\text{nd}}(b') = 2^{\text{nd}}(b) \wedge 2^{\text{nd}}(b'') = 2^{\text{nd}}(b)\\
& \quad \wedge\, 3^{\text{rd}}(b'') = \big(3^{\text{rd}}(b') \setminus p_i\big) \wedge p_i \in 3^{\text{rd}}(b')\\
& \quad \wedge\, (\text{Lift}(\mathit{RBCDeliver}^{\downarrow}))(R(u-1), R(u), p_i)\\
\mathit{RBCAgreement}(R) \triangleq{}& \forall p_i \in \text{Correct}(R) \,.\, \forall t \in \mathbb{T} \,.\, \forall b \in \mathbf{B}_{R(t-1)} \,.\\
& (\text{Lift}(\mathit{RBCDeliver}^{\downarrow}))(R(t-1), R(t), p_i) \wedge t > 0 \;\Rightarrow\\
& \forall p_j \in \text{Correct}(R) \,.\, \exists u \in \mathbb{T} \,.\, u > 0 \wedge \exists b' \in \mathbf{B}_{R(u-1)} \,.\, \exists b'' \in \mathbf{B}_{R(u)} \,.\\
& \quad \wedge\, 1^{\text{st}}(b') = 1^{\text{st}}(b) \wedge 1^{\text{st}}(b'') = 1^{\text{st}}(b)\\
& \quad \wedge\, 2^{\text{nd}}(b') = 2^{\text{nd}}(b) \wedge 2^{\text{nd}}(b'') = 2^{\text{nd}}(b)\\
& \quad \wedge\, 3^{\text{rd}}(b'') = \big(3^{\text{rd}}(b') \setminus p_j\big) \wedge p_j \in 3^{\text{rd}}(b')\\
& \quad \wedge\, (\text{Lift}(\mathit{RBCDeliver}^{\downarrow}))(R(u-1), R(u), p_j)\\
\mathit{ReliableBroadcast}(R) \triangleq{}& (\mathit{RBCValidity}(R) \wedge \mathit{RBCAgreement}(R))\\
\mathit{OmegaProperty}(R) \triangleq{}& \big(\exists p_k \in \text{Correct}(R) \,.\, \exists t \in \mathbb{T} \,.\, p_k \in \mathbf{TI}_{R(t)}\big)\\
\mathit{CorrectPBound}(R) \triangleq{}& (2 \cdot |\mathit{Correct}(R)| \geq (N+1))\\
\mathcal{F}_{\text{ad}} \triangleq{}& \{\, \text{QPPReliability},\ \mathit{ReliableBroadcast},\ \mathit{OmegaProperty},\ \mathit{CorrectPBound} \,\}
\end{aligned}$$

Figure 5.14.: Fairness assumptions for the $\Diamond\mathcal{S}$-algorithm

Figure 5.15 shows the formal definition of the $\Diamond\mathcal{S}$-algorithm based on the definitions before. We define the set of process-actions $\mathbf{\Phi}_{\text{ct}}$ as the set that contains the lifted versions of the non-lifted actions defined before. The set of events $\mathbf{\Psi}_{\text{ct}}$ contains only the two defined events *QPPSnd* and *QPPDeliver*.

We conclude with the tuple $(\ \mathrm{Init}_{\mathrm{ct}},\ \Phi_{\mathrm{ct}},\ \Psi_{\mathrm{ct}},\ \mathcal{F}_{\mathrm{ad}},\ \mathrm{C2LV}_{\mathrm{ct}},\ \mathrm{LV2C}_{\mathrm{ct}}\ )$ that defines the formal Distributed Algorithm $\mathfrak{A}_{\mathrm{ct}}$ (see figure 5.15).

$$\begin{aligned}
\Phi_{\mathrm{ct}} &\triangleq \Big\{\ \mathrm{DLift}(\mathit{While}^{\downarrow}),\ \mathrm{DLift}(\mathit{Phs1}^{\downarrow}),\ \mathrm{DLift}(\mathit{Phs2}^{\downarrow}),\ \mathrm{DLift}(\mathit{Phs3Suspect}^{\downarrow})\ \Big\}\\
&\quad \cup \Big\{\ \mathrm{DLift}(\mathit{Phs4Fail}^{\downarrow}),\ \mathrm{DLift}(\mathit{Phs4Success}^{\downarrow}),\ \mathrm{DLift}(\mathit{Crash}^{\downarrow})\ \Big\}\\
&\quad \cup \Big\{\ \mathrm{DLift}(\mathit{ImmoTrust}^{\downarrow}),\ \mathrm{Lift}(\mathit{Phs3Trust}^{\downarrow}),\ \mathrm{Lift}(\mathit{RBCDeliver}^{\downarrow})\ \Big\}\\
\Psi_{\mathrm{ct}} &\triangleq \{\ \mathit{QPPSnd},\ \mathit{QPPDeliver}\ \}
\end{aligned}$$

$$\mathfrak{A}_{\mathrm{ct}} \triangleq (\ \mathrm{Init}_{\mathrm{ct}},\ \Phi_{\mathrm{ct}},\ \Psi_{\mathrm{ct}},\ \mathcal{F}_{\mathrm{ad}},\ \mathrm{C2LV}_{\mathrm{ct}},\ \mathrm{LV2C}_{\mathrm{ct}}\ )$$

Figure 5.15.: Formal model of the $\Diamond\mathcal{S}$-algorithm

5.2.3. Proof Issues

The $\Diamond\mathcal{S}$-algorithm also is a Consensus algorithm. Therefore, again, the requirements defined in Section 2.4.1 have to be satisfied. Figure 5.16 depicts the formal assertions for the properties. Our proofs are mainly based on the lemmas and theorems given in [FMN07], but there is also a mentionable addition to the properties that are to be proven: [FMN07] gives no proof for Irrevocability. Indeed, in their work, processes that have once decided, upon do not change their values and this fact is only mentioned as a small remark without a further proof. Actually, the formulated Agreement property in [FMN07] does not rule out the case in which processes change values simultaneously from one configuration to the next. On the other hand, the structure of the proofs for the remaining properties, Validity, Agreement, and Termination, can be adopted to our Isabelle formalization almost one to one. It is again a question of the level of detail that makes a difference in using the theorem prover. Many small invariants had to be added, which in a paper proof seem to be to obvious to be mentioned. This especially concerns proofs that refer to the transition rules in [FMN07]. As already explained, a proof that states that only rules $\mathcal{A}_1, \ldots, \mathcal{A}_k$ change a variable in a configuration and, hence, no other rules have to be regarded must use the means introduced in Sections 3.1 and 3.2 explicitly. As an example of the

$$
\begin{aligned}
\text{Validity}_{\text{ct}}(R) &\triangleq \forall t \in \mathbb{T} \,.\, \forall p_i \in \mathbb{P} \,.\, dc \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \Rightarrow \big(\exists p_k \in \mathbb{P} \,.\, \text{v}_{\text{inp}}(p_k) = val \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_i)\big) \\
\text{Agreement}_{\text{ct}}(R) &\triangleq \forall t \in \mathbb{T} \,.\, \forall p_i, p_j \in \mathbb{P} \,.\, dc \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \wedge dc \triangleright \mathbf{S}_{R(t)}(p_j) \neq \bot \\
&\quad \Rightarrow val \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_i) = val \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_j) \\
\text{Termination}_{\text{ct}}(R) &\triangleq \forall p_i \in \text{Correct}(R) \,.\, \exists t \in \mathbb{T} \,.\, dc \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \\
\text{Irrevocability}_{\text{ct}}(R) &\triangleq \forall t \in \mathbb{T} \,.\, \forall p_i \in \mathbb{P} \,.\, dc \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \\
&\quad \Rightarrow \big(\forall t' \in \mathbb{T} \,.\, \big(t' \geq t \Rightarrow val \triangleright dc \triangleright \mathbf{S}_{R(t')}(p_i) = val \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_i)\big)\big)
\end{aligned}
$$

Figure 5.16.: Formal properties of Distributed Consensus for the $\Diamond\mathcal{S}$-algorithm

different level of detail that is necessary for a proof in a theorem prover consider the statement that a process in phase P2 and P4 is always the coordinator of its current round. This fact is usually regarded to be *obvious* by the pseudo code, transition rules, and actions. Although this is assumed as a fact in [FMN07], we have to strictly adhere to the given semantics in a theorem prover. Hence, we have to consider the whole code/model and establish the above fact as an invariant that must be proven by induction (see Section 3.2). Further examples of statements that seem to be obvious facts but require more than a line of proof code in a theorem prover are:

- The receiver of P1 and P3 messages is always the coordinator of the corresponding round.
- The sender of P2 and P4 messages is always the coordinator of the corresponding round.
- *PhsX* actions are only executed once per round.
- The intersection of two sets that contain more than the half of all processes, is not empty.

- If a coordinator p_c obtained more than $N/2$ messages for phase PX in round r, then more than the half of all processes have actually responded to the coordinators request because a process sends only one message per phase and round and hence the senders of the messages p_c received are mutually distinct.
- many more ...

Like Fuzzati et. al [FMN07], we strengthened the definition of Validity and proved the conjunction of the following assertions by induction:

Validity 1 Every value occuring in a broadcast is an input value of some process.

Validity 2 Every value occuring in a P1 message is an input of some process.

Validity 3 Every value occuring in a P2 message is an input of some process.

Validity 4 Every value occuring in belief of a process is an input of some process.

Validity 5 Every value occuring in decision value of a process is an input of some process.

Validity can easily be shown by induction because initially in every run the belief is set to the input value of the process, the decision value is $\perp$, and there are no broadcasts and messages. Hence, the conjunction is trivially true initially. For the inductional step we assume that all values in messages and in the process-state (beliefs and decision values) are input values. Since actions only copy values from one of these stores to another, the conjunction remains true for every step of the algorithm (the whole proof is given in $\hookrightarrow$ Isabelle[8]).

To prove Irrevocability, we applied the techniques introduced in Section 3.2 and used Theorem 3.2.0.12. Hence, to show Irrevocability it suffices to show Step, which is to show the assertion

$$\forall t' \in \mathbb{T} \,.\, \big(dc \triangleright \mathbf{S}_{R(t')}(p_i) \neq \perp\big) \Rightarrow dc \triangleright \mathbf{S}_{R(t')}(p_i) = dc \triangleright \mathbf{S}_{R(t'+1)}(p_i).$$

(If the decision value of a process p_i is not undefined then every step from t' to $t'+1$ does not change the decision value of p_i). This can be shown by case distinction on actions and events from (fixed) t' to $t'+1$. Application of the introduced concepts in Section 3.1 is used to reduce the cases to consider (the proof is given in $\hookrightarrow$ Isabelle[9]).

Obviously, the Agreement property must be shown by considering the values that are broadcasted. To show Agreement, we have to prove that broadcasted values are always *locked* values (see the previous section and [FMN07]). Then we have to show that if two values v and v' are locked, then $v = v'$ (the whole proof is given in $\hookrightarrow$ Isabelle[10]). Inspired by the proof sketches in [CT96], Fuzzati et al. [FMN07] use induction on the round number to prove this lemma: To prove a proposition P holds for all rounds r' with $r' \geq r$, the first step is to show P holds for $r = r'$. In the inductive step, P is shown for round k under the assumption that P holds for all r' with $r \leq r' < k$. For a detailed description of this proof and the proof of Termination we refer the reader to [FMN07]. Regarding the timeline, this approach dissents from standard

[8]Isabelle/HOL theory: `CTValidity.thy`, Lemma(s): `Validity`
[9]Isabelle/HOL theory: `CTAgreement.thy`, Lemma(s): `Irrevocability`
[10]Isabelle/HOL theory: `CTAgreement.thy`, Lemma(s): `LockingAgreement, Agreement`

temporal reasoning techniques, as the 'global' round number does not proceed consistently with the global clock. In fact, the round number might be different in all local states of the processes and can evolve independently from the global progress as long as it is monotonically increasing; it is possible that a round number r_i of process p_i is greater than the round number r_j of a process j and later in time $r_i < r_j$ holds. Therefore, proofs done by this technique are intricate as well as hard to follow by a reader, and therefore are not chosen for doing formal proofs. There infeasibility can be highlighted by errors that we found in [FMN07, Fuz08]. Making such errors within a theorem proving environment is not possible and, hence, we were forced to correct them [KNR12].

For the proof of Termination, we had to show that every correct process eventually decides a value. The concept of the proof is again very close to the reasoning done by Fuzzati et. al [FMN07]. As explained by Fuzzati et al., we proved a lemma *Infinity* ($\hookrightarrow$ Isabelle[11]) to show that in every infinite run R and for every round number $r > 0$ there is a process $p_i \in \mathbb{P}$ such that p_i is in round number r and phase P3 at some time t in R. It turns out that the reasoning in [FMN07] is very hard to formalize in a theorem prover. Fuzzati et. al. give an upper bound for the maximum number of steps a process may perform with a fixed program counter ($r_t,\ P_t$). In Isabelle, we had to define a function that represents that number of steps and we had to show that the assertions of [FMN07] concerning the concrete values of this functions are correct. This means much more effort than the arguments given in [FMN07]. Hence, to deduce that the program counter must be incremented after the given number of derivations, we had to spend about 3k lines of proof code only to prove this first lemma of Fuzzati's proof for Termination.

After showing this fundamental lemma, the rest of the proof is done more or less analogously to [FMN07]. We were able to show that all Admissible Runs for the $\Diamond\mathcal{S}$-algorithm are finite ($\hookrightarrow$ Isabelle[12]) and to prove that a process can only be deadlocked under certain defined conditions ($\hookrightarrow$ Isabelle[13]). This finally enabled us to prove Termination the way as it works in [FMN07].

Broadcastfree Variant of $\Diamond\mathcal{S}$

We demonstrated the reusability of our approach in a master thesis which examinded a variant of this algorithm without broadcast. Although the student was not familiar with Isabelle, after some introduction to the theoretical premises of the previous work, he was able to verify the modified algorithm by using and modifying the provided lemmas and theorems. It turned out that whereas verifying Validity, Agreement, and Irrevocability was rather simple, but the proof of Termination was more complex requiring additional fairness assumptions.

There are implicit modeled fairness assumptions in the model of [FMN07], which do not apply in the case without broadcast. To explain these implicit assumptions, let us assume there is a broadcast for a process p_i available that has to be delivered. Since processes deliver a broadcast message and decide simultaneously a respective value in one atomic step, and, furthermore, the *ReliableBroadcast* assumption ensures that every broadcast message is eventually delivered, it is implied that p_i eventually decides. In our modified variant we replaced the broadcast message

[11] Isabelle/HOL theory: `CTProofTermination.thy`, Lemma(s): `Infinity`
[12] Isabelle/HOL theory: `CTProofTermination.thy`, Lemma(s): `Finiteness_of_Admissible_Runs`
[13] Isabelle/HOL theory: `CTProofTermination.thy`, Lemma(s): `processDeadlock1,processDeadlock2`

by a point-to-point message. Hence, let us assume that in this scenario we have a point-to-point message for process p_i available carrying the value to decide. Due to the QPPReliability assumption, this message has to be received eventually by process p_i. In the case of point-to-point messages we use the standard way of delivering and receiving messages that has been proposed in Section 2.3.5. As also proposed in [FMN07] this implies that the receipt of the message and the step of processing the message are autonomic atomic actions. Hence, in this case, the existing fairness assumption QPPReliability does not imply that the message that has been sent to make a process p_i decide upon a value is actually processed by p_i. Therefore, we had to introduce weak fairness for the *PhsX* (*Phs1*$^{\downarrow}$, *Phs2*$^{\downarrow}$, *Phs3Trust*$^{\downarrow}$, *Phs3Suspect*$^{\downarrow}$, *Phs4Success*$^{\downarrow}$, *Phs4Fail*$^{\downarrow}$) actions in this setting to prove Termination.

5.3. Paxos

The Paxos algorithm has been introduced by Leslie Lamport in [Lam98]. In his original work Lamport described the algorithm in terms of a parliament of an ancient Greek island named Paxos. Later, for the purpose of understanding, Lamport shortened his description of the algorithm [Lam01] and (together with Eli Gafni) introduced some variant of the algorithm that used shared disks, instead of message passing in [GL00]. Although Paxos is supposed to solve the problem of Distributed Consensus, strictly speaking, it is not a Consensus algorithm because there is no proof of Termination for this algorithm. [Fuz08] argues that Paxos does not terminate even under 'best conditions' and proved a theorem *Non Termination*, which states that in their model of the Paxos algorithm, there can always be an infinite run. Regarding the impossibility result of [FLP85], it appears that without other assumptions, Paxos is not able to solve Consensus in an asynchronous setting with crash failures. Nevertheless Lamport states, that the Paxos algorithm terminates if the system becomes stable for a sufficiently long time. De Prisco et al. [PLL97] claim that they have given a definition, which can capture the conditions under which Paxos satisfies Termination.

Since our interest lies on providing means to do proofs for distributed algorithms in theorem proving environments but does not concern the analysis of the ability of certain algorithms to solve distributed problems, we have only considered the properties that have been inspected in the original paper, and, hence, in this case the safety properties (Validity, Agreement, and Irrevocability) for this algorithm.

The Paxos algorithm has many similarities to the $\Diamond\mathcal{S}$-algorithm considered in Section 5.2. Regarding the message protocols given in Figure 5.17 we observe that in the Paxos algorithm there is a *leader*, paralleling the coordinator in the $\Diamond\mathcal{S}$-algorithm. In an asynchronous round r of the Paxos algorithm this leader starts a round by sending the request number r to all processes in $\mathbb{P}$. As [Fuz08] we call such a message a P0 message. This message is needed because the Paxos algorithm does not use the same rotating coordinator paradigm as the algorithms in the previous sections, i.e., a leader process in the Paxos algorithm can skip arbitrary many rounds when it starts a new round. Therefore, the P0 message is needed to establish a new round number. P1 and P2 messages are identical for both algorithms: processes send their current beliefs to the coordinator/leader and the coordinator/leader selects the best belief (the belief with the most recent stamp) and sends it back to the processes. For both algorithms a majority of messages is required before the selection can be made and for both algorithms the coordinator/leader requires, furthermore, a majority of P3 messages to be sure that the round has succeeded and to prepare for the decision. Since we keep our model close to the formalizations in [Fuz08], we also provided the broadcast message for both algorithms even if in Lamport's original work there is no broadcast mentioned. Fuzzati has introduced the broadcast for the Paxos algorithm with the argument that 'it is mandatory that all the other correct processes receive the same *success* message'.

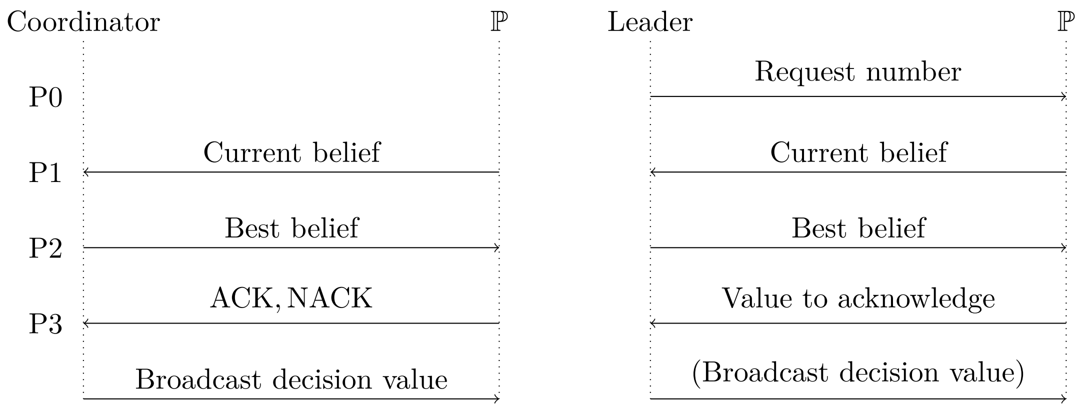

Figure 5.17.: Round message protocol $\Diamond\mathcal{S}$-algorithm vs. Paxos

Although Paxos is very similar to the $\Diamond\mathcal{S}$-algorithm, there are some essential differences relating to assumptions on the setting and, especially, the used failure model.

In contrast to the $\Diamond\mathcal{S}$-algorithm, the Paxos algorithm works without the assumption of a Quasi-reliable point-to-point communication, i.e., the failure model of Paxos allows message loss. Moreover, it also allows the duplication of messages.

Processes can crash (as for the $\Diamond\mathcal{S}$-algorithm), but may recover. Since processes may recover, but do not have to, the chosen scenario also includes the case where processes do not recover. Hence, the assumptions are strictly weaker for the Paxos algorithm than the assumptions for the $\Diamond\mathcal{S}$-algorithm. Paxos assumes stable storage that is used in the case of recovery. In this storage, data is persistently stored and will therefore not be lost in case of process crash.

5.3.1. Informal Introduction to the Algorithm

In [Lam98] the Paxos algorithm is explained not by means of pseudo code but, more or less, in natural language through many pages of the paper (maybe that is why Lamport often states that people find it hard to understand). Lamport gives a more formal and more precise description in the appendix. To give a short intuitive representation of the algorithm Fuzzati [Fuz08] rewrote Paxos, i.e., what a process does in a round, in six steps. Since our model is very close to the model of Fuzzati, the pseudo code we present here is based on these six steps using the terms of Fuzzati rather than the terms of Lamport's original paper. Lamport [Lam98] proposed three roles a process can play: proposer, acceptor, and learner. Processes acting as proposers (or leaders) act similar to the coordinator in the $\Diamond\mathcal{S}$-algorithm and, hence, start new requests (indexed by numbers similar to the round numbers for the $\Diamond\mathcal{S}$-algorithm), collect messages and disseminate beliefs. Acceptors reply to the messages of proposers and answer to the requests and proposals of the proposer. Finally, learners use the value the proposer sends in a *success* broadcast to make decisions.

Four variables that are used by the algorithm are assumed to be permanently stored (to a persistent storage) only to be updated but never to be deleted (see above):

- r: A variable which stores the last request number that a process has proposed.
- p: A variable which stores the last request number to which a process answered as an acceptor.
- belief : The current belief of the process (see Section 5.2).
- decision : The decision value of the process.

As in the $\Diamond\mathcal{S}$-algorithm, the two latter variables are pairs: a value (belonging to the set of possible decision values) and a request number. Additionally, we allow the decision to be $\perp$ to represent an undecided state. Listing 5.3 shows the initial setup for these variables.

Listing 5.3: Initialization for a participant of the Paxos algorithm

```
r              := 0
p              := 0
belief.val     := Input
belief.stamp   := 0
decision       := ⊥
```

The pseudo code in listing 5.4 shows the behaviour of a proposer. At first, a proposer p_i chooses a new request number r. For the algorithm it is important that different processes use different request numbers. If we assume processes to be numbered $p_1, \ldots, p_N$ then this can be satisfied if every process p_i uses only numbers $r > 0$ such that $(r \mod N) = i$. Moreover, request numbers chosen by a process p_i must increase. Therefore, we use a function nxreq(r,p_i) for the choice of the next request number and assume that this function returns a value greater than r from the request numbers of p_i. After setting the value of r to the new request number, p_i sends the chosen number r to [all] participants (for simplicity we assume that p_i sends the number to all participants, although Lamport explains that it would suffice to send messages to a majority). As in the $\Diamond\mathcal{S}$-algorithm, p_i awaits replies from a majority of the participants containing their current beliefs. From these beliefs p_i selects the best (the one with the most recent stamp), stores it to a local variable b, and sends it back to the participants as a proposal. If, again, a majority of the processes answers by sending a message to p_i then p_i initiates the decision process by broadcasting b.

Listing 5.4: Paxos algorithm for proposer/leader p_i

```
r := nxreq(r,p_i)
for 1 ≤ j ≤ $N$
    SEND(p_i, (r, P0), r) TO p_j
await |received.$P1^r$| ≥ $\lceil \frac{N+1}{2} \rceil$
    b := best(received.$P1^r$)
for 1 ≤ j ≤ $N$
    SEND(p_i, (r, P2), b) TO p_j
await |received.$P3^r$| ≥ $\lceil \frac{N+1}{2} \rceil$
BROADCAST(p_i,b)
```

The corresponding behaviour of an acceptor p_j is given in Listing 5.5. An acceptor p_j first waits until it receives a P0 message containing a request number r with r greater than its own value p(which is the last request number of a request to which p_j has answered). If p_j received such a message from a process p_i then p_j sends its current belief belief in a P1 message to p_i and waits until p_i again answers with a corresponding P2 message with the best belief. p_j adopts the value of the best belief to its own belief value and sets the stamp to p. Then, as acknowledgement, p_j returns the value in a P3 message to p_i.

Listing 5.5: Paxos algorithm for acceptor p_j

```
await received.$P0^{rq} \neq \emptyset$ for some rq ≥ p from some process p_i
p := rq
SEND(p_j, (p,P1), belief) TO p_i
await received.$P2^p \neq \emptyset$
if (received.$P2^p$ = {(p_i,(p,P2),b)}) then {
    belief.val   := b.val
    belief.stamp := p
   SEND(p_j, (p,P3),b.val) TO p_i
}
```

The learner role of our Paxos pseudo code equals the RBC_DELIVER procedure in the $\Diamond\mathcal{S}$-algorithm: if a process p_i receives a broadcast message, then it decides for the respective value.

Listing 5.6: Paxos algorithm for learner p_i

```
when (RBC_DELIVER (p_j, d) {
  if (decision = ⊥) then {
    decision := d
    output(decision.value)
  }
}
```

To be non-blocking, processes are allowed to skip waiting for messages, to declare themselves as leaders, and to start a new round at any point of the algorithm. Hence, semantics of the used **await** statement may be interpreted differently than the corresponding statement in the $\Diamond\mathcal{S}$-algorithm.

At any point in time a process may crash and recover. Following a crash, all received messages of the process are deleted as are the messages in its outbox. Consequently, following recovery, the mentioned four variables, which have been stored persistently, are the only local data that survive a crash.

5.3.2. Formal Model

The required datastructures for configurations, Localviews, and messages for the Paxos algorithm resemble the datastructures for the $\Diamond\mathcal{S}$-algorithm. Concepts for the datastructures are mainly influenced by the definitions of [Fuz08]. A configuration for our $\Diamond\mathcal{S}$-algorithm has four entries:

- **S**: the array of process states
- **Q**: the Message History
- **BufIn**: an array of inboxes for the processes
- **B**: the set of broadcast messages

Compared to the formal models introduced before, the configuration entry for the inboxes is new. It is required because a process p_i, p_i is only able to access the messages it received since it has recovered from its last crash. Therefore, messages p_i received are inserted into the respective buffer $\mathbf{BufIn}(p_i)$. $\mathbf{BufIn}(p_i)$ is then emptied everytime p_i recovers from a crash.

Again, we denote the array of process states of a configuration $\mathfrak{C}$ by $\mathbf{S}_{\mathfrak{C}}$, the inbox of a process p_i in $\mathfrak{C}$ by $\mathbf{BufIn}_{\mathfrak{C}}(p_i)$ and the set of broadcast messages by $\mathbf{B}_{\mathfrak{C}}$. For Paxos we use the Message History with message loss as introduced in Chapter 2.3.5. For broadcasts, we use the modeling techniques introduced in Section 2.3.6 but since there can be only one broadcast per request number, we use request numbers as indices for the broadcasts. Hence, we model the broadcast as a function over request numbers (which are natural numbers).

The state of a process p_i is represented by a tuple $(\ (\ l,\ P\ ),\ r,\ p,\ (\ v,\ s\ ),\ (\ c,\ \iota\ ),\ d\ )$, where (c.f. [Fuz08])

- l is a boolean value, which is true if and only if p_i is leader and $P \in \{$ Idle, Trying, Waiting, Polling, Done $\}$ is a phase label.
- r is the last request number p_i has chosen.
- p is the last request number p_i has answered to as an acceptor (the *prepare request number*).
- $(\ v,\ s\ )$ is the belief of p_i consisting of a value $v \in \mathbb{I}$ and a stamp $s \in \mathbb{N}$ (see section 5.2.1).
- c is a boolean value that is true if and only if p_i is alive (not crashed) and ι is the incarnation number of p_i. The incarnation number counts how often a process recovered from a crash. This number is used for message administration: processes are allowed to send only messages that originate from the same incarnation.
- d is the decision value of p_i. d must be either $\perp$ (undecided) or a pair $(\ v_d,\ s_d\ )$ where v_d is the value p_i decided and s_d is the round number of the round where the decision was taken.

Again, we use the introduced means for notation to denote the entries of a process state $\mathbf{S}_{\mathfrak{C}}(p_i)$ of a process p_i in a configuration $\mathfrak{C}$:

- the pair $(\ l,\ P\ )$ of p_i is denoted by $a \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the last request number of p_i is denoted by $rnd \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.
- the prepare request number of p_i is denoted by $p \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$
- the belief of p_i is denoted by $b \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$, the value of the belief of p_i is denoted by $val \triangleright b \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$, and the stamp of the belief of p_i is denoted by $stamp \triangleright b \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.
- the $(\ c,\ \iota\ )$ pair of p_i is denoted by $g \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$, the c value of the pair is denoted by $isUp \triangleright g \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$, and the ι value is denoted by $ic \triangleright g \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.
- the decision value of p_i is denoted by $dc \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$, the value of the decision of p_i is denoted by $val \triangleright dc \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$, and the stamp of the belief of p_i is denoted by $stamp \triangleright dc \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.

As for the $\diamond\mathcal{S}$-algorithm, the Localview of p_i comprises p_i's process state, the inbox $\mathbf{In}_{lv}$ of p_i (the set of received messages of p_i), a set $\mathbf{Out}_{lv}$ for the outbox of the process, an entry $\mathbf{B}_{lv}$ for the Broadcast History, and the identity $\mathbf{ID}_{lv}$ of p_i. In a Localview lv a process can access its process state $\mathbf{S}_{lv}$ and the respective entries $a \triangleright \mathbf{S}_{lv}$, $rnd \triangleright \mathbf{S}_{lv}$, $p \triangleright \mathbf{S}_{lv}$, $b \triangleright \mathbf{S}_{lv}$, $val \triangleright b \triangleright \mathbf{S}_{lv}$, $stamp \triangleright b \triangleright \mathbf{S}_{lv}$, $g \triangleright \mathbf{S}_{lv}$, $isUp \triangleright g \triangleright \mathbf{S}_{lv}$, $ic \triangleright g \triangleright \mathbf{S}_{lv}$, and $dc \triangleright \mathbf{S}_{lv}$.

Again, we expect the values of the inbox and the identity to be read-only values and we model the outbox as an emptyset where p_i puts in the messages p_i is willing to send (as already explained and used in sections 2.3.2 and 2.3.5).

In this algorithm we use tuples $(\ s,\ r,\ req,\ phs,\ c,\ ic\ )$ for messages where

- $s \in \mathbb{P}$ is the sender of m.
- $r \in \mathbb{P}$ is the receiver of m.
- $req \in \mathbb{N}$ is the request number the message is associated with.
- $phs \in \{\ \mathrm{P0},\ \mathrm{P1},\ \mathrm{P2},\ \mathrm{P3}\ \}$ is the phase the message is associated with.
- c is an arbitrary content of the message.
- ic is the incarnation of s when the message has been put into the outbox of s.

The sender of a message m is denoted by snd_m, the receiver by rcv_m, the round by rnd_m, the phase by P_m, the content of a message is denoted by cnt_m, and the respective incarnation number is denoted by ic_m.

Initially, there is no process with a leader role and every process p_i starts in phase Idle with request number and prepare request number set to 0. Furthermore, the belief is set to $(\ \mathrm{v_{inp}},\ 0\ )$, p_i is not crashed, and the incarnation number counts zero incarnations. p_i is initially undecided. Furthermore, the Message History contains no messages and hence returns zero for every pair of messages and message status values. Since there are no broadcasts in the initial Broadcast History but the Broadcast History is a (total) function over request numbers we map all request numbers to $\bot$ initially. Initially, the inbox for every process is empty. Therefore, we define the initial process state array $\mathbf{S}_0$, the initial Message History $\mathbf{C}_0$, the array of inboxes $\mathbf{BufIn}_0(p_i)$, the Broadcast History $\mathbf{B}_0$, and the set of initial states $\mathrm{Init_{ct}}$ as depicted in Figure 5.18.

Analogous to [Fuz08], we divide the Paxos algorithm in four phases for the leader:

$$\mathbf{S}_0(p_i) \triangleq (\ (\ \text{False},\ \text{Idle}\ ),\ 0,\ 0,\ (\ \text{v}_{\text{inp}}(p_i),\ 0\ ),\ (\ \text{True},\ 1\ ),\ \perp\ ) \qquad \mathbf{C}_0(m,t) \triangleq 0$$

$$\mathbf{BufIn}_0(p_i) \triangleq \emptyset \qquad \mathbf{B}_0(r) \triangleq \perp$$

$$\text{Init}_{\text{px}} \triangleq \left\{ \begin{pmatrix} \mathbf{S}_0 \\ \mathbf{C}_0 \\ \mathbf{BufIn}_0 \\ \mathbf{B}_0 \end{pmatrix} \right\}$$

Figure 5.18.: Initial definitions for the Paxos Algorithm

- Idle: At the beginning a process is Idle.
- Trying: If a process decides to declare himself as a leader and to start a new round, it transits from Idle to Trying.
- Waiting: When a leader process sends the P1 message, it transits from phase Trying to the phase Waiting (it waits for the replies of the participants).
- Polling: After having received a majority of P1 replies, a leader process sends the P2 message and transits from phase Waiting to phase Polling.
- Done: A successful leader that has received a majority of P3 broadcasts a success message and transits from phase Polling to Done.

As stated before, a process is always able to start a new request or end its leader role. Furthermore, a process is always able to answer to requests (independently from the phase it is in).

Again, we model crashes by providing a DnA ***Crash***$^{\downarrow}$ (see Figure 5.19), which is always enabled while the process is not already crashed and has not yet decided (decided processes are assumed to be terminated). In the case of crash-failure $isUp \triangleright g \triangleright \mathbf{S}_{lv}$ for the process is set to False.

No other action but ***Recovery***$^{\downarrow}$ is enabled for a process with $\neg isUp \triangleright g \triangleright \mathbf{S}_{lv}$. *Recovery*$^{\downarrow}$ increments the incarnation number of the process, empties its inbox and sets $isUp \triangleright g \triangleright \mathbf{S}_{lv}$ back to True. Furthermore, the phase of the process is set back to Idle and the process starts again as being not in a leader role. Of course, this action preserves the values for the four persistently stored variables $rnd \triangleright \mathbf{S}_{lv}$, $p \triangleright \mathbf{S}_{lv}$, $b \triangleright \mathbf{S}_{lv}$ and $dc \triangleright \mathbf{S}_{lv}$.

The DnA ***LeaderYes***$^{\downarrow}$ enables a process p_i to assume the leader role for a new request number. This request number is chosen from p_i's set of request numbers $\text{ReqNo}(p_i)$ by function $\text{nxreq}(p_i, r)$. We use the Hilbert choice operator 'ϵ' to express that $\text{nxreq}(p_i, r)$ returns some value r' with $r' \in \text{ReqNo}(p_i)$ and $r' > r$ without any further restrictions on this value. *LeaderYes*$^{\downarrow}$ is only enabled for process p_i, if p_i is not already in a leader role and is in phase Idle. The boolean value for being a leader is then set to True by *LeaderYes*$^{\downarrow}$.

$$\begin{aligned}
\mathit{Crash}^{\downarrow}_{en}(lv) &\triangleq \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
&\quad \wedge dc \triangleright \mathbf{S}_{lv} = \bot \\
\mathit{Crash}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ a \triangleright \mathbf{S}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ (\ \text{False},\ ic \triangleright g \triangleright \mathbf{S}_{lv}\ ),\ dc \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv} \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{Crash}^{\downarrow} &\triangleq (\ \mathit{Crash}^{\downarrow}_{en},\ \mathit{Crash}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

$$\begin{aligned}
\mathit{rcIncarn}(lv) &\triangleq (\ \text{True},\ ic \triangleright g \triangleright \mathbf{S}_{lv} + 1\ ) \\
\mathit{Recovery}^{\downarrow}_{en}(lv) &\triangleq \neg \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
&\quad \wedge dc \triangleright \mathbf{S}_{lv} = \bot \\
\mathit{Recovery}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ (\ \text{False},\ \text{Idle}\ ),\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ \mathit{rcIncarn}(lv),\ dc \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv}[i := \emptyset] \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{Recovery}^{\downarrow} &\triangleq (\ \mathit{Recovery}^{\downarrow}_{en},\ \mathit{Recovery}^{\downarrow}_{\Delta}\ )
\end{aligned}$$

Figure 5.19.: Actions $\mathit{Crash}^{\downarrow}$ and $\mathit{Recovery}^{\downarrow}$

The dual DnA $\boldsymbol{\mathit{LeaderNo}}^{\downarrow}$ models the inverse step from being a leader to not being a leader anymore. Therefore, this step is only possible for a process that has a leader role. $\mathit{LeaderNo}^{\downarrow}$ then sets the respective boolean variable to False and sets the phase to Idle.

In every phase, a leader is able to start a new request by the DnA $\boldsymbol{\mathit{New}}^{\downarrow}$ a leader. The new request number is chosen by the function $\text{nxreq}(p_i, r)$ and the phase is set to Trying.

If a leader p_i executes DnA $\boldsymbol{\mathit{SndPrepReq}}^{\downarrow}$ (see Figure 5.21), p_i must be in phase Trying. This action puts a set of P0 messages into the outbox of the process and sets p_i's phase to Waiting.

An acceptor p_j answers to some P0 message by executing $\boldsymbol{\mathit{RcvPrepReq}}^{\downarrow}$. This action (and the later actions $\mathit{RcvAccReq}^{\downarrow}$ and $\mathit{Success}^{\downarrow}$) is modeled as a non-deterministic action because the received message that is processed is chosen non-deterministically (similar as described for the Rotating Coordinator algorithm). p_j is only supposed to answer to a P0 message if the request number of the message is greater than the current value of p_j's variable $p \triangleright \mathbf{S}_{lv}$. Then $\mathit{RcvPrepReq}^{\downarrow}$ generates a correspondent P1 message to answer to this message containing p_j's current belief.

If a leader p_i in phase Waiting received enough P1 messages containing beliefs of the participants, it selects the *best* belief and sends it back to the participants in a P2 message by executing the DnA $\boldsymbol{\mathit{SndAccReq}}^{\downarrow}$. To chose the *best* belief p_i makes use of the functions highest_{lv}, winner_{lv} and best_{lv}. Similar to the $\Diamond\mathcal{S}$-algorithm these functions determine the belief with the most recent (*highest*) stamp. After the execution of $\mathit{SndAccReq}^{\downarrow}$, p_i's phase is Polling.

$$
\begin{aligned}
\mathit{LeaderYes}^{\downarrow}_{en}(lv) &\triangleq \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
&\quad \wedge dc \triangleright \mathbf{S}_{lv} = \bot \\
&\quad \wedge a \triangleright \mathbf{S}_{lv} = (\ \text{False},\ \text{Idle}\ ) \\
\text{nxreq}(p_i, r) &\triangleq \epsilon r'.\ r' \in \text{ReqNo}(p_i) \wedge r < r' \\
\mathit{LeaderYes}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ (\ \text{True},\ \text{Idle}\ ),\ \text{nxreq}(\mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv}),\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ dc \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv} \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{LeaderYes}^{\downarrow} &\triangleq (\ \mathit{LeaderYes}^{\downarrow}_{en},\ \mathit{LeaderYes}^{\downarrow}_{\Delta}\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{LeaderNo}^{\downarrow}_{en}(lv) &\triangleq \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
&\quad \wedge dc \triangleright \mathbf{S}_{lv} = \bot \\
&\quad \wedge (\exists P\ .\ a \triangleright \mathbf{S}_{lv} = (\ \text{True},\ P\ )) \\
\mathit{LeaderNo}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ (\ \text{False},\ \text{Idle}\ ),\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ dc \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv} \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{LeaderNo}^{\downarrow} &\triangleq (\ \mathit{LeaderNo}^{\downarrow}_{en},\ \mathit{LeaderNo}^{\downarrow}_{\Delta}\ )
\end{aligned}
$$

$$
\begin{aligned}
\mathit{New}^{\downarrow}_{en}(lv) &\triangleq \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
&\quad \wedge dc \triangleright \mathbf{S}_{lv} = \bot \\
&\quad \wedge (\exists P\ .\ a \triangleright \mathbf{S}_{lv} = (\ \text{True},\ P\ )) \\
\mathit{New}^{\downarrow}_{\Delta}(lv) &\triangleq \begin{pmatrix} (\ (\ \text{True},\ \text{Trying}\ ),\ \text{nxreq}(\mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv}),\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ dc \triangleright \mathbf{S}_{lv}\ ) \\ \mathbf{In}_{lv} \\ \mathbf{Out}_{lv} \\ \mathbf{B}_{lv} \\ \mathbf{ID}_{lv} \end{pmatrix} \\
\mathit{New}^{\downarrow} &\triangleq (\ \mathit{New}^{\downarrow}_{en},\ \mathit{New}^{\downarrow}_{\Delta}\ )
\end{aligned}
$$

Figure 5.20.: Actions $\mathit{LeaderYes}^{\downarrow}$, $\mathit{LeaderNo}^{\downarrow}$ and $\mathit{New}^{\downarrow}$

$$\begin{aligned}
\mathit{SndPrepReq}^{\downarrow}_{en}(lv) &\triangleq \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv}\\
&\wedge a \triangleright \mathbf{S}_{lv} = (\text{ True, Trying })
\end{aligned}$$

$$\mathit{SndPrepReq}^{\downarrow}_{\Delta}(lv) \triangleq \left(\begin{array}{c} (\ (\text{ True, Waiting }),\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ dc \triangleright \mathbf{S}_{lv}\ )\\ \mathbf{In}_{lv}\\ \{\ (\ \mathbf{ID}_{lv},\ p_j,\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \text{P0},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ ic \triangleright g \triangleright \mathbf{S}_{lv}\ )\ \mid\ p_j \in \mathbb{P}\ \}\\ \mathbf{B}_{lv}\\ \mathbf{ID}_{lv} \end{array}\right)$$

$$\mathit{SndPrepReq}^{\downarrow} \triangleq (\ \mathit{SndPrepReq}^{\downarrow}_{en},\ \mathit{SndPrepReq}^{\downarrow}_{\Delta}\ )$$

$$\begin{aligned}
\mathit{RcvPrepReq}^{\downarrow}(lv, lv') &\triangleq \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv}\\
&\wedge \mathbf{B}_{lv'} = \mathbf{B}_{lv}\\
&\wedge \mathbf{In}_{lv'} = \mathbf{In}_{lv}\\
&\wedge (\exists m \in \mathbf{In}_{lv}\ .\ p \triangleright \mathbf{S}_{lv} < \text{cnt}_m\\
&\wedge \text{P}_m = \text{P0}\\
&\wedge \text{rnd}_m = \text{cnt}_m\\
&\wedge \mathbf{S}_{lv'} = (\ a \triangleright \mathbf{S}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \text{cnt}_m,\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ dc \triangleright \mathbf{S}_{lv}\ )\\
&\wedge \mathbf{Out}_{lv'} = \{\ (\ \mathbf{ID}_{lv},\ \text{snd}_m,\ \text{cnt}_m,\ \text{P1},\ b \triangleright \mathbf{S}_{lv},\ ic \triangleright g \triangleright \mathbf{S}_{lv}\ )\ \})
\end{aligned}$$

$$\begin{aligned}
\mathit{SndAccReq}^{\downarrow}_{en}(lv) &\triangleq \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv}\\
&\wedge a \triangleright \mathbf{S}_{lv} = (\text{ True, Waiting })\\
&\wedge |\{\ m \in \mathbf{In}_{lv}\ \mid\ \exists c, ic\ .\ \exists p_j \in \mathbb{P}\ .\ m = (\ p_j,\ \mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \text{P1},\ c,\ ic\ )\ \}| > \frac{N}{2}
\end{aligned}$$

$$\begin{aligned}
\text{highest}_{lv} &\triangleq \text{Max}\{\ s\ \mid\ \big(\exists m \in \mathbf{In}_{lv}\ .\ \text{rnd}_m = \mathit{rnd} \triangleright \mathbf{S}_{lv} \wedge \text{P}_m = \text{P1} \wedge \text{snd}_m = p_i \wedge 2^{\text{nd}}\,(\text{cnt}_m) = s\big)\ \}\\
\text{winner}_{lv} &\triangleq \mathit{Min}\{\ i\ \mid\ \big(\exists m \in \mathbf{In}_{lv}\ .\ \text{rnd}_m = \mathit{rnd} \triangleright \mathbf{S}_{lv} \wedge \text{P}_m = \text{P1} \wedge \text{snd}_m = p_i \wedge 2^{\text{nd}}\,(\text{cnt}_m) = \text{highest}_{lv}\big)\ \}\\
\text{best}_{lv} &\triangleq \epsilon b.\ (\exists m \in \mathbf{In}_{lv}\ .\ \text{rnd}_m = \mathit{rnd} \triangleright \mathbf{S}_{lv} \wedge \text{P}_m = \text{P1} \wedge \text{snd}_m = p_{\text{winner}_{lv}} \wedge b = \text{cnt}_m)
\end{aligned}$$

$$\mathit{SndAccReq}^{\downarrow}_{\Delta}(lv) \triangleq \left(\begin{array}{c} (\ (\text{ True, Polling }),\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ dc \triangleright \mathbf{S}_{lv}\ )\\ \mathbf{In}_{lv}\\ \{\ (\ \mathbf{ID}_{lv},\ p_j,\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \text{P2},\ \text{best}_{lv},\ ic \triangleright g \triangleright \mathbf{S}_{lv}\ )\ \mid\ p_j \in \mathbb{P}\ \}\\ \mathbf{B}_{lv}\\ \mathbf{ID}_{lv} \end{array}\right)$$

$$\mathit{SndAccReq}^{\downarrow} \triangleq (\ \mathit{SndAccReq}^{\downarrow}_{en},\ \mathit{SndAccReq}^{\downarrow}_{\Delta}\ )$$

Figure 5.21.: Actions $\mathit{SndPrepReq}^{\downarrow}$, $\mathit{RcvPrepReq}^{\downarrow}$ and $\mathit{SndAccReq}^{\downarrow}$

If an acceptor p_j receives a P2 message for a request number equal to p_j's value of $p \triangleright \mathbf{S}_{lv}$, then p_j can reply to this message by executing the action ***RcvAccReq***$^{\downarrow}$ (see Figure 5.22). *RcvAccReq*$^{\downarrow}$ puts a P3 message as an acknowledgement into the outbox of p_j. Furthermore, the belief of p_j is updated with the belief value from the P2 message and stamp $p \triangleright \mathbf{S}_{lv}$.

If a leader p_i is successful and receives a majority of P3 messages for his request, then p_i can execute ***Success***$^{\downarrow}$ to initiate the decision process. By executing *Success*$^{\downarrow}$ p_i broadcasts the value it received in the P3 messages (which is the value it has formerly sent in the P2 message) as the decision value and transits to phase Done.

A broadcast is finally processed by the action ***RbcDeliver***$^{\downarrow}$. The process p_i that executes *RbcDeliver*$^{\downarrow}$ adopts the value from the broadcast as its decision value (if it has not already decided a value).

Similar as for the algorithms described before, we define two actions ***ChSnd*** and ***ChDeliver*** to manage message movements from one message status to the next (see Section 2.3.5). *ChSnd* shifts an **outgoing** message to the messages with status **transit** and *ChDeliver* advances a message from **transit** to status **received** (see Figure 5.23). These two events use the introduced subactions Transmit and Receive (see Section 2.3.5). As stated before, processes are only allowed to send messages that have been generated within the same incarnation. Therefore for Paxos there is an additional condition for the *ChSnd* event: $\mathrm{ic}_m = ic \triangleright g \triangleright \mathbf{S}_{\mathfrak{C}}(\mathrm{snd}_m)$. The *ChDeliver* event also updates the array of inboxes $\mathbf{BufIn}_{\mathfrak{C}'}$ by putting the delivered message into the inbox of the receiver. As for the algorithm considered before, we decided to model Transmit and Receive as events.

In the Paxos setting also message loss and message duplication is possible. Therefore, we provide two additional events ***ChDupl*** and ***ChLose***. *ChDupl* uses the introduced subaction Duplicate to create another copy of a message in **transit**. The subaction Lose is used by *ChLose* to shift a message from status **transit** to **lost** (see Section 2.3.5).

Figure 5.24 shows the final definitions for the formal Paxos algorithm. Functions $\mathrm{C2LV}_{\mathrm{px}}$ and $\mathrm{LV2C}_{\mathrm{px}}$ are used to convert configurations to Localviews and vice versa. The function $\mathrm{C2LV}_{\mathrm{px}}$ generates a Localview out of a configuration $\mathfrak{C}$ for a process p_i by returning a vector of the

- state of a process: $\mathbf{S}_{\mathfrak{C}}(p_i)$
- the inbox of p_i, which is determined as the entry for p_i in the array of inboxes in $\mathfrak{C}$
- an emptyset for the outbox
- the Broadcast History
- the identity of p_i: p_i

$\mathrm{LV2C}_{\mathrm{px}}$ embeds a Localview lv back into a given configuration $\mathfrak{C}$ by

- setting the state of process p_i to the process state $\mathbf{S}_{lv}$ of the Localview
- updating the **outgoing** messages by adding the messages from p_i's outbox (note that sndupd is the function defined in Section 2.3.5)

$$
\begin{aligned}
\mathit{RcvAccReq}^{\downarrow}_{en}(lv, lv') \triangleq\ & \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
& \wedge \mathit{stamp} \triangleright b \triangleright \mathbf{S}_{lv} < p \triangleright \mathbf{S}_{lv} \\
& \wedge \mathbf{B}_{lv'} = \mathbf{B}_{lv} \\
& \wedge \mathbf{In}_{lv'} = \mathbf{In}_{lv} \\
& \wedge (\exists m \in \mathbf{In}_{lv} \ .\ p \triangleright \mathbf{S}_{lv} = \mathrm{rnd}_m \\
& \wedge \mathrm{P}_m = \mathrm{P2} \\
& \wedge \mathbf{Out}_{lv'} = \big\{ \ (\ \mathbf{ID}_{lv},\ \mathrm{snd}_m,\ p \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ 1^{\mathrm{st}}\,(\mathrm{cnt}_m),\ \mathit{ic} \triangleright g \triangleright \mathbf{S}_{lv}\ ) \ \big\} \\
& \wedge \mathbf{S}_{lv'} = (\ a \triangleright \mathbf{S}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ (\ 1^{\mathrm{st}}\,(\mathrm{cnt}_m),\ p \triangleright \mathbf{S}_{lv}\ ),\ g \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ ))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Success}^{\downarrow}(lv, lv') \triangleq\ & \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
& \wedge a \triangleright \mathbf{S}_{lv} = (\ \mathrm{True},\ \mathrm{Polling}\ ) \\
& \wedge |\{\ m \in \mathbf{In}_{lv} \ \mid\ \exists c, \mathit{ic}\ .\ \exists p_j \in \mathbb{P}\ .\ m = (\ p_j,\ \mathbf{ID}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \mathrm{P3},\ c,\ \mathit{ic}\ )\ \}| > \frac{N}{2} \\
& \wedge \mathbf{In}_{lv'} = \mathbf{In}_{lv} \\
& \wedge (\exists m \in \mathbf{In}_{lv}\ .\ \mathrm{rnd}_m = \mathit{rnd} \triangleright \mathbf{S}_{lv} \\
& \wedge \mathrm{P}_m = \mathrm{P3} \\
& \wedge \mathbf{B}_{lv'} = \mathbf{B}_{lv}[\mathit{rnd} \triangleright \mathbf{S}_{lv} := (\ \mathrm{cnt}_m,\ \mathbb{P}\ )] \\
& \wedge \mathbf{Out}_{lv'} = \mathbf{Out}_{lv} \\
& \wedge \mathbf{S}_{lv'} = (\ (\ \mathrm{True},\ \mathrm{Done}\ ),\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ \mathit{dc} \triangleright \mathbf{S}_{lv}\ ))
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{sdc}(v, r, d) \triangleq\ & \begin{cases} d & \text{, if } d \neq \bot \\ (v, r) & \text{, else} \end{cases} \\
\mathit{RbcDeliver}^{\downarrow}(lv, lv') \triangleq\ & \mathit{isUp} \triangleright g \triangleright \mathbf{S}_{lv} \\
& \wedge \mathbf{In}_{lv'} = \mathbf{In}_{lv} \\
& \wedge \mathbf{Out}_{lv'} = \mathbf{Out}_{lv} \\
& \wedge (\exists r\ .\ \mathbf{B}(r) \neq \bot \\
& \wedge \mathbf{ID}_{lv} \in 2^{\mathrm{nd}}\,(\mathbf{B}(r)) \\
& \wedge \mathbf{B}_{lv'} = \mathbf{B}_{lv}[r := (\ 1^{\mathrm{st}}\,(\mathbf{B}(r)),\ 2^{\mathrm{nd}}\,(\mathbf{B}(r)) \setminus \mathbf{ID}_{lv}\ )] \\
& \wedge \mathbf{S}_{lv'} = (\ a \triangleright \mathbf{S}_{lv},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ p \triangleright \mathbf{S}_{lv},\ b \triangleright \mathbf{S}_{lv},\ g \triangleright \mathbf{S}_{lv},\ \mathrm{sdc}(1^{\mathrm{st}}\,(\mathbf{B}(r)),\ r,\ \mathit{dc} \triangleright \mathbf{S}_{lv})\ ))
\end{aligned}
$$

Figure 5.22.: Actions $\mathit{RcvAccReq}^{\downarrow}$, $\mathit{Success}^{\downarrow}$ and $\mathit{RbcDeliver}^{\downarrow}$

$$\begin{aligned}
\mathit{ChSnd}(\mathfrak{C},\mathfrak{C}') \triangleq\ & \mathbf{B}_{\mathfrak{C}'} = \mathbf{B}_{\mathfrak{C}} \\
& \wedge \mathbf{BufIn}_{\mathfrak{C}'} = \mathbf{BufIn}_{\mathfrak{C}} \\
& \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}} \\
& \wedge (\exists m \ .\ \mathit{isUp} \rhd g \rhd \mathbf{S}_{\mathfrak{C}}(\mathrm{snd}_m) \\
& \wedge \mathrm{ic}_m = \mathit{ic} \rhd g \rhd \mathbf{S}_{\mathfrak{C}}(\mathrm{snd}_m) \\
& \wedge \mathrm{Transmit}(\mathbf{C}_{\mathfrak{C}}, \mathbf{C}_{\mathfrak{C}'}, m))
\end{aligned}$$

$$\begin{aligned}
\mathit{ChDeliver}(\mathfrak{C},\mathfrak{C}') \triangleq\ & \mathbf{B}_{\mathfrak{C}'} = \mathbf{B}_{\mathfrak{C}} \\
& \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}} \\
& \wedge (\exists m \ .\ \mathit{isUp} \rhd g \rhd \mathbf{S}_{\mathfrak{C}}(\mathrm{rcv}_m) \\
& \wedge \mathbf{BufIn}_{\mathfrak{C}'} = \mathbf{BufIn}_{\mathfrak{C}}[\mathrm{rcv}_m := \mathbf{BufIn}_{\mathfrak{C}}(\mathrm{rcv}_m) \cup \{\ m\ \}] \\
& \wedge \mathrm{Receive}(\mathbf{C}_{\mathfrak{C}}, \mathbf{C}_{\mathfrak{C}'}, m))
\end{aligned}$$

$$\begin{aligned}
\mathit{ChDupl}(\mathfrak{C},\mathfrak{C}') \triangleq\ & \mathbf{B}_{\mathfrak{C}'} = \mathbf{B}_{\mathfrak{C}} \\
& \wedge \mathbf{BufIn}_{\mathfrak{C}'} = \mathbf{BufIn}_{\mathfrak{C}} \\
& \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}} \\
& \wedge (\exists m \ .\ \mathrm{Duplicate}(\mathbf{C}_{\mathfrak{C}}, \mathbf{C}_{\mathfrak{C}'}, m))
\end{aligned}$$

$$\begin{aligned}
\mathit{ChLose}(\mathfrak{C},\mathfrak{C}') \triangleq\ & \mathbf{B}_{\mathfrak{C}'} = \mathbf{B}_{\mathfrak{C}} \\
& \wedge \mathbf{BufIn}_{\mathfrak{C}'} = \mathbf{BufIn}_{\mathfrak{C}} \\
& \wedge \mathbf{S}_{\mathfrak{C}'} = \mathbf{S}_{\mathfrak{C}} \\
& \wedge (\exists m \ .\ \mathrm{Lose}(\mathbf{C}_{\mathfrak{C}}, \mathbf{C}_{\mathfrak{C}'}, m))
\end{aligned}$$

Figure 5.23.: Events *ChSnd*, *ChDeliver*, *ChDupl* and *ChLose*

- updating the array of inboxes in the configuration by the inbox of p_i in the Localview (this is necessary because the process-action $\mathit{Recovery}^{\downarrow}$ empties the inbox)
- adopting the local Broadcast History to the global

We define the set of process-actions $\mathbf{\Phi}_{\text{px}}$ as the set that contains the lifted versions of the non-lifted actions defined before. The set of events $\mathbf{\Psi}_{\text{px}}$ contains the four defined events *ChSnd*, *ChDeliver*, *ChDupl*, and *ChLose*. For the Paxos algorithm we are only interested in safety. Therefore, we require no further fairness assumptions.

We conclude with the tuple $(\ \text{Init}_{\text{px}},\ \Phi_{\text{px}},\ \Psi_{\text{px}},\ \emptyset,\ \text{C2LV}_{\text{px}},\ \text{LV2C}_{\text{px}}\ )$ that defines the formal Distributed Algorithm $\mathfrak{A}_{\text{px}}$.

$$\text{C2LV}_{\text{px}}(\mathfrak{C}, p_i) \triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}(p_i), \\ \mathbf{BufIn}_{\mathfrak{C}}(p_i), \\ \emptyset, \\ \mathbf{B}_{\mathfrak{C}}, \\ p_i \end{pmatrix}$$

$$\text{LV2C}_{\text{px}}(lv, p_i, \mathfrak{C}) \triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{lv}] \\ \text{sndupd}_{\mathbf{C}_{\mathfrak{C}}, \mathbf{Out}_{lv}} \\ \mathbf{BufIn}_{\mathfrak{C}}[p_i := \mathbf{In}_{lv}] \\ \mathbf{B}_{lv} \end{pmatrix}$$

$$\begin{aligned} \Phi_{\text{px}} \triangleq & \Big\{ \text{DLift}(\mathit{Crash}^{\downarrow}),\ \text{DLift}(\mathit{Recovery}^{\downarrow}),\ \text{DLift}(\mathit{LeaderYes}^{\downarrow}),\ \text{DLift}(\mathit{LeaderNo}^{\downarrow}) \Big\} \\ & \cup \Big\{ \text{DLift}(\mathit{New}^{\downarrow}),\ \text{DLift}(\mathit{SndPrepReq}^{\downarrow}),\ \text{DLift}(\mathit{SndAccReq}^{\downarrow}) \Big\} \\ & \cup \Big\{ \text{Lift}(\mathit{RcvPrepReq}^{\downarrow}),\ \text{Lift}(\mathit{RcvAccReq}^{\downarrow}),\ \text{Lift}(\mathit{Success}^{\downarrow}),\ \text{Lift}(\mathit{RbcDeliver}^{\downarrow}) \Big\} \\ \Psi_{\text{px}} \triangleq & \{\ \mathit{ChSnd},\ \mathit{ChDeliver},\ \mathit{ChDupl},\ \mathit{ChLose}\ \} \\ \mathfrak{A}_{\text{px}} \triangleq & (\ \text{Init}_{\text{px}},\ \Phi_{\text{px}},\ \Psi_{\text{px}},\ \emptyset,\ \text{C2LV}_{\text{px}},\ \text{LV2C}_{\text{px}}\ ) \end{aligned}$$

Figure 5.24.: Formal Rotating Coordinator Algorithm

5.3.3. Proof Issues

We formally proved all required safety properties for our Isabelle/HOL model of the Paxos algorithm: Validity, Agreement, and Irrevocability (as defined in Section 2.4.1, see Figure 5.25). As in Lamport's original work, we do not provide a formal proof for Termination. As for the

$$\begin{aligned}
\text{Validity}_{\text{ct}}(R) &\triangleq \forall t \in \mathbb{T} \,.\, \forall p_i \in \mathbb{P} \,.\, dc \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \Rightarrow \big(\exists p_k \in \mathbb{P} \,.\, \mathrm{v_{inp}}(p_k) = \mathit{val} \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_i)\big) \\
\text{Agreement}_{\text{ct}}(R) &\triangleq \forall t \in \mathbb{T} \,.\, \forall p_i, p_j \in \mathbb{P} \,.\, dc \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \wedge dc \triangleright \mathbf{S}_{R(t)}(p_j) \neq \bot \\
&\quad \Rightarrow \mathit{val} \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_i) = \mathit{val} \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_j) \\
\text{Irrevocability}_{\text{ct}}(R) &\triangleq \forall t \in \mathbb{T} \,.\, \forall p_i \in \mathbb{P} \,.\, dc \triangleright \mathbf{S}_{R(t)}(p_i) \neq \bot \\
&\quad \Rightarrow \big(\forall t' \in \mathbb{T} \,.\, \big(t' \geq t \Rightarrow \mathit{val} \triangleright dc \triangleright \mathbf{S}_{R(t')}(p_i) = \mathit{val} \triangleright dc \triangleright \mathbf{S}_{R(t)}(p_i)\big)\big)
\end{aligned}$$

Figure 5.25.: Formal properties of Distributed Consensus for the Paxos algorithm

$\Diamond\mathcal{S}$-algorithm, proofs are based on the work of Fuzzati [Fuz08], but we have added a proof for the property Irrevocability. Proofs mainly rely on simple invariants that are shown by application of the methods introduced in Sections 3.1 and 3.2. Validity is strengthened analogously to Section 5.2.3 and is the conjunction of the following assertions:

Validity 1 Every value occuring in a broadcast is an input value of some process.

Validity 2&3 Every value occuring in a P1 or P2 message is an input of some process.

Validity 2&3a Every value occuring in a P3 message is an input of some process.

Validity 4 Every value occuring in belief of a process is an input of some process.

Validity 5 Every value occuring in decision value of a process is an input of some process.

Note that Validity 2&3a is (erroneously) missing in [Fuz08]. In our view it is required for the later induction because the formal model (our formal model and the one used in [Fuz08]) makes explicit use of the value of some P3 message for the broadcast in the $\mathit{Success}^{\downarrow}$ action. Therefore the induction does not work without regarding also the value of P3 messages.

The proof is analogously to that in Section 5.2.3: initially, in every run the belief is set to the input value of the process, the decision value is $\bot$, and there are no broadcasts and messages. Hence, initially the conjunction is trivially true. For the inductional step we assume that there are only input values in all messages, belief, and decision values. Since actions only copy values from one of those locations to another, the conjunction remains true for every step of the algorithm (the formal proof is given in $\hookrightarrow$ Isabelle[14]).

To prove Irrevocability the theorem from the $\Diamond\mathcal{S}$-algorithm could be reused requiring for only minor modifications concerning the names of the variables in configurations ($\hookrightarrow$ Isabelle[15]).

[14]Isabelle/HOL theory: `PxValidity.thy`, Lemma(s): `Validity`

[15]Isabelle/HOL theory: `PxAgreement.thy`, Lemma(s): `Irrevocability`

Likewise the proof of Agreement is similar to the respective proof for the $\Diamond\mathcal{S}$-algorithm (our complete proof is given in $\hookrightarrow$ Isabelle[16]). It uses the same notion of *locked* values. A detailed description of this proof can be found in [Fuz08]. During our work, we encountered several problems in the model and in the proofs. The major problems were found in the proofs for Paxos of [Fuz08] (as we already noted in [KNR12]) and are:

- There was an error in the broadcast mechanism that circumvented a delivery of broadcasts to all processes except for its sender, rendering executions, in circumstances in which only a minimal majority of processes were alive when the first process decides, nonterminating. Moreover, an assumption about the mechanism claimed that every broadcast will eventually be received by all correct processes. Due to the error mentioned before, this is in contradiction to the transition rules. Of course, from this contradiction one could derive any property needed.
- Another problem concerned the basic orderings that are introduced for the reasoning on process states. It turned out that the ordering does not fullfill the required monotoncity in time that was assumed. Since many proofs for the following lemmas relied on this ordering, this problem is serious.
- The proof for one of the central lemmas (Locking Agreement) is wrong. It uses another lemma (Proposal Creation), but its assumptions are not fullfilled. Therefore, we had to find an adequate version of this lemma with weaker assumptions and redo both proofs (a similar error occurs in [FMN07]).

5.4. Using the Alpha-Abstraction

Our last case study deals with a more general framework for Consensus algorithms. The framework is introduced by Guerraoui and Raynal in [GR07]. In their work, they define an abstraction *Alpha* which captures Consensus safety.

In our case study, we focus on the implementation of the *Alpha* abstraction in a shared memory system. First, we are going to introduce the general ideas of Guerraoui et. al. ([GR07]).

In addition to the *Alpha* abstraction, the framework makes use of the more commonly known *Omega* abstraction, which can be considered as an abstraction for *eventual leader detection* and is also known as the weakest failure detector to solve Consensus [CHT96]. More specifically Guerraoui and Raynal introduce a boolean function Omega(). Invoked by a process p_i, Omega() returns True if and only if the process is elected as leader. Omega() is assumed to satisfy the following property *Eventual Leadership* [GR07]:

> There is a time τ and a correct process p_i such that, after τ, every invocation of Omega() by p_i returns True, and any invocation of Omega() by $p_j \neq p_i$ returns False.

This guarantees the eventual election of an unique and correct leader. As Guerraoui states, the *Alpha* abstraction was motivated by the objective to devise the complement of *Omega*[GR07].

[16]Isabelle/HOL theory: `PxAgreement.thy`, Lemma(s): `Agreement`

While *Omega* captures Consensus liveness, the *Alpha* abstraction is supposed to capture Consensus safety by introducing a 'shared one-shot storage object', which is invoked by processes with a value to store. If two or more processes access the object concurrently, the object does not guarantee that values are stored. In this case the object remains with its initial value $\perp$. But if processes access the object sequentially, the first process that accesses the object will succeed in storing its value and the object will keep this value permanently [GR07]. Process access the object by invoking a function Alpha. Alpha uses two parameters: a round number r and the value to store v. It returns the stored value. As for Paxos, it is assumed that (1) processes use distinct round numbers and (2) that each process p_i uses strictly increasing round numbers [GR07]. Guerraoui and Raynal showed that in the context of their framework, Distributed Consensus is solved if the function Alpha satisfies the following four properties[GR07]:

α**Validity:** If the invocation Alpha(r, v) returns, the returned value is either $\perp$ or a value v' such that there is a round $r' \leq r$, and Alpha(r', v') has been invoked by some process.

α**Quasi-agreement:** For each two invocations Alpha(r, x) and Alpha(r', y) that return v and v' respectively, the following assertion must hold:

$$(v \neq \perp \wedge v' \neq \perp) \Rightarrow (v = v') \, .$$

α**Conditional non-$\perp$ convergence:** An invocation $I = \text{Alpha}(r, x)$ must return a non-$\perp$ value if every invocation $I' = \text{Alpha}(r', y)$ that starts before I returns is such that $r' \leq r$.

α**Termination:** Any invocation of Alpha by a correct process returns.

Note that Guerraoui and Raynal wrote '$r' < r$' in property αConditional non-$\perp$ convergence (instead of '$r' \leq r$') but obviously the assertion

Every invocation $I' = \text{Alpha}(r', y)$ that starts before I returns is such that $r' < r$

formally can never be true since $I = \text{Alpha}(r, x)$ starts before I returns and $r \not< r$. Therefore, we have chosen to write $r' \leq r$.

Listing 5.7 depicts the application of abstractions *Alpha* and *Omega*. The framework solves the problem of Distributed Consensus if Omega satisfies the *Eventual Leadership* property and Alpha satisfies αValidity, αQuasi-agreement, αConditional non-$\perp$ convergence, and αTermination in the presence of up to $N - 1$ crash failures. In this setting, we assume that crashed processes do not recover.

To calculate a decision value, a participant p_i of the algorithm executes the function Consensus(v) (see Listing 5.7) where v denotes the input value of p_i. During the execution, p_i uses a variable r for the round number (note that the round number of participant p_i is actually r+i) and a variable res to store the result of invocations of Alpha. The variable DECIDED is a shared atomic variable, which is used to disseminate a decision value (analogously to the broadcasts in Paxos and Chandra/Toueg's algorithm). Initially, r is set to 0. While p_i is not decided, the process checks continuously if it is the leader by using function Omega(). If this check succeeds p_i invokes Alpha(r+i, v) and stores the returned value to variable res. If this value is $\perp$, p_i advances to the next round by incrementing r (note that r is incremented by N to ensure that processes use distinct round numbers). If the result is a value different from $\perp$ then p_i sets the shared variable DECIDED to this value.

The desired safety properties are satisfied by the utilisation of the assumed properties of Alpha and the *Eventual Leadership* property is used to guarantee Termination. Let us at first consider the Agreement property. Processes will not output different values: if processes p_i and p_j output values v_i and v_j, then the shared variable DECIDED must have been set to v_i and v_j at some time t_i and t_j. Due to the while-condition (see Listing 5.7) the value of DECIDED must have been different from $\perp$ before it is output by p_i (p_j respectively). Moreover, since DECIDED is set to a value different from $\perp$ only if it is set to a return value of Alpha, at the time when p_i (p_j) evaluated the while-condition, DECIDED must have been set to such a return value v_i' (v_j') of Alpha with $v_i' \neq \perp$ ($v_j' \neq \perp$). Since DECIDED is a shared variable, of course, other processes are able to manipulate DECIDED while p_i (p_j) advances from the while-loop to the output statement. But due to the if-clause in line 7 and 8 once set to $v \neq \perp$, it can never be set back to $\perp$. Moreover, since the value of res is always set to the return value of Alpha by the αQuasi-agreement property, every process has the same return value v (or $\perp$) for res, and, hence, DECIDED is always set to the same value v. Therefore we obtain $v = v_i = v_i' = v_j' = v_j$. This implies Agreement and Irrevocability.

Listing 5.7: Generic framework for Consensus, code for process p_i [GR07]

```
 1  function consensus(v):
 2  r := 0
 3  while (DECIDED = ⊥)
 4  {
 5     if Omega() then {
 6        res := Alpha(r+i, v);
 7        if (res ≠⊥) then
 8           DECIDED := res
 9        else
10           r := r + N
11     }
12  }
13  output(DECIDED)
```

Listing 5.7 shows that function Alpha is always invoked with the input value of the process. As already stated, an assignment of $\perp$ to the variable DECIDED is ruled out by the if-clause in line 7 and 8. Hence, an output value must be a return value $v \neq \perp$ of Alpha. With αValidity this implies Validity.

For the verification of the Termination property, assume for contradiction that there is a correct process that does not decide a value. By inspection of the code (and with Irrevocability) this can only be the case when we have DECIDED$= \perp$ forever. This implies that all processes, including the eventual leader p_c, do not decide. Since Omega() eventually returns True only for p_c, after some time τ, only p_c increments its round number and invokes Alpha. Therefore, eventually p_c will reach a round number r greater than all round numbers that were used by other processes for invocations of Alpha. Hence, for the next invocation I of Alpha by p_c the assertion that *every invocation* $I' = \text{Alpha}(r', y)$ *that starts before* I *returns is such that* $r' \leq r$ will be true. By αConditional non-$\perp$ convergence this implies that I returns a non-$\perp$ value and

DECIDED will be set to this value. Hence, DECIDED can not be $\perp$ forever: a contradiction.

To derive a formal, proof we modeled the given pseudo code similarly to the algorithms considered in the previous chapters and finally showed the following theorem:

Theorem 5.4.0.1 (α-Consensus)
Every run R of the framework satisfies

- Validity(R)
- Agreement(R)
- Irrevocability(R)
- Termination(R)

Proof. $\hookrightarrow$ Isabelle[17]

To understand the formal model it is necessary to understand that invocations of Alpha must not be modeled as atomic steps. Regarding the given pseudo code we notice that communication is completely abstracted away by Alpha (and Omega), i.e. communication between the processes is (obviously) required to implement functions Alpha and Omega such that they satisfy the required properties, but is hidden by these abstractions. Although it is not necessary for the proof of Theorem 5.4.0.1 to know the implementation of Alpha and Omega, the communication that takes place during an invocation of Alpha makes it indispensable to model invocations I of Alpha as fixed periods in a run R such that they start at some configuration $R(t)$ and end in $R(t+t_I)$. In our model we defined an action `InvokeAlpha` for the start and an action `AlphaReturn` for the end of an invocation. Of course, other processes were allowed to execute actions and also invoke *Alpha* in between. The introduction of `InvokeAlpha` and `AlphaReturn` enabled us to define the four assumptions on *Alpha* properly so that it could be demonstrated that these assumptions imply that the framework represents a family of Consensus algorithms.

The reason why this framework actually provides a family of Consensus algorithms is that Alpha can be implemented for arbitrary communication mechanisms. In [GR07] Guerraoui and Raynal present implementations of the *Alpha* abstraction using (1) Shared Memory, (2) disks, (3) message passing and (4) active disks.

Since in this case study we want to consider an algorithm using shared memory for communication, we focussed on (1), and, hence this last case study deals with the implementation of *Alpha* in a shared memory system as it is described by Guerraoui and Raynal (see Section 4 in [GR07]). The proposed algorithm is similar to an algorithm proposed by Gafni for his *Round-by-round failure detectors* (see Section 4.2 in [Gaf98]). Gafni's algorithm was originally not intended to solve Consensus but to implement the adopt-commit protocol in a shared memory system. The algorithm presented by Guerraoui and Raynal can be seen as an adaption of Gafni's algorithm for Distributed Consensus. As written by Guerraoui and Raynal, the algorithm can also be interpreted as a variant of the Paxos algorithm (see [Lam98], [GL00]).

[17] Isabelle/HOL theory: `Prototype.thy`, theorem(s): `AdmissibleImpliesValidity, AdmissibleImpliesAgreement, Irrevocability, Termination`

The assumed communication mechanism employed by Guerraoui and Raynal is that of Regular Registers as introduced in Section 2.3.7. Let us recall the informal description of the properties of a Regular Register (for a formal description see Section 2.3.7):

- a read not concurrent with a write gets the correct value,
- a read that overlaps a write obtains either the old or new value,

We assume that our system has N Regular Registers $\mathrm{Reg}_{p_1}, \ldots, \mathrm{Reg}_{p_N}$. Moreover, only process p_i is allowed to write to register Reg_{p_i} but Reg_{p_i} can be read by any process. Hence, we speak of 1WNR Regular Registers (1 writer, N readers). Moreover, we assume the registers to be *reliable*, i.e., registers do not crash and do not corrupt data.

Every register stores a tuple (*lre*, *lrww*, *val*). The entries *lre* and *lrww* are used to store round numbers and *val* stores either an input value or ⊥.

The next section explains how these registers can be used to solve Consensus.

5.4.1. Informal Introduction to the Algorithm

The implementation of the *Alpha* abstraction is given by Guerraoui and Raynal [GR07] as a piece of pseudo code that simply shows the computation steps for the function Alpha with arguments r and v. The pseudo code as it is presented in [GR07] is given in Figure 5.8.

Listing 5.8: Alpha in a shared memory system [GR07]

```
function Alpha(r,v): // This is the text for p_i
// [Step 1] ————————————————————————————————
// 1.1: p_i first makes public the date of its last attempt
REG[i].lre := r
// 1.2: Then, p_i reads the shared registers to know the other processes progress
reg[1..N] := REG[1..N] // p_i reads (in any order) the regular registers
// 1.3: p_i aborts its attempt if another process started a higher round
if (∃j. reg[j].lre > r)) then return (⊥)
// [Step 2] ————————————————————————————————
// Then p_i adopts the last value that has been deposited in a register
// If there is no such value, it adopts its own value v
let value be reg[j].val where j is such that ∀k. reg[j].lrww ≥ reg[k].lrww
if (value = ⊥) then value := v
// [Step 3] ————————————————————————————————
// 3.1: p_i writes the value it has adopted (together with the current date)
REG[i].(lrww, v) := (r,value)
// 3.2: p_i reads again the shared registers to know the processes' progress
reg[1..N] := REG[1..N]
if (∃j. reg[j].lre > r) then return (⊥)
// [Step 4] ————————————————————————————————
// Otherwise, value is the result of the Alpha abstraction: p_i returns it
return (value)
```

We provide the original pseudo code from [GR07] with all its details and comments, on the one hand, to give the reader an intuition for the algorithm and on the other hand to exemplify how the latent ambiguity of pseudo code can lead to typical pitfalls, when transferring a given pseudo code to a formal model. With this example we want to raise the readers sensitivity for how typical concurrency problems might be hidden by the kind of presentation, and, in this case, by the pseudo code.

The pseudo code uses record notation to denote access to the registers. Hence, REG[i]. lre denotes the entry for the lre field of register Reg_{p_i}. Each process holds its own copy of the registers in a local variable reg, which is repeatedly updated during an invocation of Alpha(see pseudo code in Figure 5.8). The same notation used for registers is used for the variable reg (reg. lre ,reg. lrww, reg. val).

As already explained, the function uses two parameters: a round number r and the value to store v. It returns the actual value stored.

In reference to the pseudo code, the function Alpha can be implemented for Regular Registers by executing the following steps:

1. (line 4) the lre entry of p_i's register is set to the current round number r.
2. (line 6) p_i updates its local copy of the registers by copying the values of the registers to its local variable reg.
3. (line 8) if p_i recognizes that there is a process p_j that has already written a higher round number to p_j's lre entry reg[j]. lre, then p_i aborts the calculation and returns $\perp$.
4. (line 12) otherwise p_i determines the process p_j with the greatest lrww entry reg[j]. lrww and sets a local variable value to reg[j]. val.
5. (line 13) this value will be the value p_i adopts for the designated return value unless it turns out to be $\perp$. In the latter case p_i sets value to its own input value v.
6. (line 16) p_i writes the current round number r to its register for entry reg[j]. lrww and the content of variable value to entry reg[j]. val.
7. (line 18) p_i updates again its local copy of the registers by copying the values of the registers to its local variable reg.
8. (line 19) if p_i recognizes that there is a process p_j that has already written a higher round number to p_j's lre entry reg[j]. lre, then p_i aborts the calculation and returns $\perp$.
9. (line 22) otherwise p_i returns the value that is stored in variable value.

Regarding the examples given in the previous sections, we recognize an obvious correspondence from atomic steps to the the lines of pseudo code. Therefore, the reader might expect that extracting a formal model from pseudo code means just to create an action for every line of pseudo code. Of course, we already have given examples where two or more lines of pseudo code can be summarized by one action, but this example shows that many steps that seem to be atomic indeed are not and moreover, have to be handled very carefully since modeling these steps as one atomic step would hide exactly the problems resulting from concurrency and more specifically from the definition of Regular Registers.

As we know from Section 2.3.7, the problematic but interesting cases for Regular Registers are those in which reads and writes to the same register are concurrent. If we modeled the first step (which is actually a write to p_i's register) as an atomic action, it would not be possible to have a concurrent read to this write. Hence, we are again forced to model this step as at least two steps: a begin of the write and an end of the write. Furthermore, it must be possible for other processes to read between or overlapping with these two actions (see Section 2.3.7). We find the same situation in step 6, concerning the write to entries reg[j].lrww and reg[j].val (note that Guerraoui and Raynal assume that processes trivially know the last value they wrote to a register and hence to update only some entries by writing the old values to the other). In steps 2 and 7 we find read operations for the registers. We recognize that there are indeed N read operations appearing in only one single line of pseudo code. Furthermore (as remarked in the comments), the order in which the registers are read, must be nondeterministicly chosen (otherwise we would only prove correctness for a special case of this algorithm). Again, for every read overlapping with writes (and other read operations) to the register must be possible. And, again, this is enabled by modeling a read by a *read begin* and a *read end* (cf. Section 2.3.7) action. Hence, for the 'execution' of this single line of pseudo code we actually obtain $2N$ executions of actions. This example illustrates the hiding of concurrency by the pseudo code representation.

For correctness the four properties αValidity, αConditional non-$\perp$ convergence, αQuasi-agreement, and αTermination (see previous section) must be shown. Intuitively, it is again the 'no values are invented' argument that implies αValidity: there are simply no other values stored in the val entries of registers and variables, and the returned value originates either from such an entry or is $\perp$.

Guerraoui and Raynal [GR07] state that the αTermination property is satisfied because the algorithm is wait-free. This argument will be reconsidered in the later sections.

The argument for αConditional non-$\perp$ convergence is again adopted from [GR07]: let us assume an invocation $I = \text{Alpha}(r, v)$ by a process p_i such that there is no invocation $I' = \text{Alpha}(r', v')$ with $r' > r$, that starts before I returns. Hence, p_i sees no greater values than its own and p_i always proceeds with the execution of Alpha. Since p_i does not prematurely abort the execution of Alpha, it must finally return the non-$\perp$ value that has been chosen in lines 12 and 13.

Our viewpoint is that a proof needs all formal details and can only be given for a formal model. Therefore, we restrict ourselves, again, to a very rough idea of the proof for αQuasi-agreement for the given pseudo code at this point and provide a full formal proof for our formal Isabelle model (see next section, a more intuitive extensive description of the proof is given by Guerraoui and Raynal [GR07]). For αQuasi-agreement, we have to consider two invocations $I = \text{Alpha}(r, v)$ and $I' = \text{Alpha}(r', v')$ that return a value different from $\perp$. Without loss of generality, let us assume $r < r'$. Obviously, both invocations found no lre values higher than there own round number in other registers for the checks in lines 8 and 19. Hence, I' can not have finished the write operation in line 4 before I began the read operations in lines 6 and 18. Hence, p_i must have finished the write operation in line 16 (because $16 < 18$) before I' started reading the registers in line 6 (because $4 < 6$). This implies that the value that I has written must have been taken into account when I' chooses the value for decision in line 12. Of course, there might be further invocations in between such that I' chooses a value from a different register. But, since round numbers are natural numbers, we can recursively use the same argumentation for the round number of the invocation from which the value is chosen and this implies αQuasi-agreement.

5.4.2. Formal Model

Although our model is supposed to examine executions of Distributed Algorithms and the pseudo code for Alpha seems to be only a description of one single function, it is possible to prove the required properties in terms of our model. In our formal setting, we consider processes, which invoke Alpha infinitely often with increasing round numbers. Hence, we get a distributed algorithm where each process sequentially performs invocations of Alpha in an infinite loop (for the algorithm we use the term α-algorithm in the following). The only assumptions we make for the invocations are the two assumptions of Guerraoui and Raynal: (1) distinct processes use distinct round numbers and (2) each process uses strictly increasing round numbers. Then we show that the desired properties hold in every admissible run and therefore in every possible setting that satisfies the requirements. Note that we do not need the formal model for the framework since the properties of Alpha are defined independently from the framework.

The datastructures for configurations and Localviews differ from the datastructures of the previous algorithms because there are no messages and therefore is no Message History but the introduced module for shared memory and Regular Registers (see Section 2.3.7).

A configuration for the α-algorithm has only two entries:

- **S**: the array of process states
- **Sh**: the module for shared memory, i.e., for the Regular Registers

Again, we denote the array of process states of a configuration $\mathfrak{C}$ by $\mathbf{S}_{\mathfrak{C}}$.

To model the Regular Registers, we use the module specified in Section 2.3.7 with

$$\mathcal{V}_{\text{reg}} = (\mathbb{N} \times \mathbb{N} \times (\mathbb{I} \cup \{ \perp \}))\,.$$

This enables us to store tuples (*lre*, *lrww*, *val*) in the registers as described in the previous section. The state of the module for shared memory of a configuration $\mathfrak{C}$ is denoted by $\mathbf{Sh}_{\mathfrak{C}}$. For access on the registers we use the introduced subactions *RdBegin*, *RdEnd*, *WriteBegin* and *WriteEnd*. Hence, there is no direct access to the entries of this module. Based on the assumption that it is trivial for a process to store the most recent values it has written to the register, analogously to Guerraoui and Raynal, we allow to update single entries of the registers. In our formal model we write $\mathit{WriteBegin}(\mathbf{Sh}, \mathbf{Sh}', p_i, (\ r',\ \text{-},\ \text{-}\ ))$ to denote that from module state **Sh** to $\mathbf{Sh}'$ process p_i performs a *WriteBegin* subaction that writes a tuple (r', x, y) to the register Reg_{p_i} where r' is a new value for the first entry and x and y are values equal to the values that have been already stored in the second and third entry of the register (these values remain unchanged).

To serialize the used actions of a process p_i for the α-algorithm we use the following phases:

- Idle: p_i is Idle before the invocation starts.
- Publishing: While p_i writes its own value to the register entry *lre* (see pseudo code line 4), p_i is in phase Publishing.

- Fetching: Before p_i starts reading one of the registers (see pseudo code line 6), p_i is in phase Fetching.
- FetchingWait: After p_i started reading one of the registers and while p_i waits for the returned value, p_i is in phase FetchingWait (see pseudo code line 6). Note that there is a loop where p_i alternately is in phase Fetching and phase FetchingWait until all registers are read.
- Proposing: When p_i starts writing the *lrww* and *val* entries of the register (see pseudo code line 16), p_i is in phase Proposing. Note that there is no communication in lines 12, 13 and therefore the begin of the write and the choice of the value can be modeled as one step. The abort in line 8 is modeled as an alternative action to the begin of the write and therefore no phases are modeled for lines 7-15.
- FetchingAgain: After p_i finished writing the *lrww* and *val* entries of the register and before p_i starts again reading a register, p_i is in phase FetchingAgain.
- FetchingAgainWait: After p_i started reading one of the registers for the second time and while p_i waits for the returned value, p_i is in phase FetchingAgainWait (see pseudo code line 18). Again there is a loop where p_i alternately is in phase FetchingAgain and phase FetchingAgainWait until all registers are read.
- Returned$_{v'}$: After p_i finally returned a value v', p_i is in phase Returned$_{v'}$. Note that we have $v' = \bot$ when p_i returns $\bot$.
- Crashed: After p_i crashed, p_i is in state Crashed.

The state of a process p_i is represented by a tuple $(\ r,\ v,\ p,\ l,\ c\ )$, where

- r is p_i's round number (the first parameter for the invocation of Alpha).
- v is p_i's value (the second parameter for the invocation of Alpha).
- p is p_i's phase in the current invocation of Alpha.
- l is a list of processes, which is used to determine the next register p_i reads. Before each read cycle this list contains all processes and the order of the processes is determined non-deterministically.
- c is p_i's local copy of the registers. This copy is updated twice per invocation (cf. pseudo code).

We denote a list with (ordered) elements $q_1, \dots, q_n$ by $[q_1, \dots, q_n]$. For a list l we define an operation hd that returns the head (the first element) of the list and an operation tl that returns the tail (the rest) of the list. $[\ ]$ denotes the empty list. $\text{hd}([\ ])$ is undefined and we define $\text{tl}([\ ]) \triangleq [\ ]$. Furthermore we assume a predicate distinct, which is true for a list l if and only if the elements in l are pairwise distinct. The function $\text{set}(l)$ returns a set of all elements of l. This definition corresponds to the lists defined in Isabelle/HOL (see [NPW02]).

Again we use the introduced means for notation to denote the entries of a process state $\mathbf{S}_{\mathfrak{C}}(p_i)$ of a process p_i in a configuration $\mathfrak{C}$:

- The round number of p_i is denoted by $rnd \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.
- p_i's value v is denoted by $v \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.
- The list of processes that is used to determine p_i's next register to read is denoted by $rtd \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.
- Finally, p_i's copy of the registers is denoted by $reg \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$.

For p_i's local copy of the registers we use the notation $lre \triangleright reg(p_i) \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$ (respectively $lrww \triangleright reg(p_i) \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$ and $val \triangleright reg(p_i) \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$) to denote the entry for the *lre* (respectively *lrww* and *val*) value of the register. Note that we do not need this notation for the entries of the registers but only for the local copies of the registers. This is due to the fact that a register is only accessed by the defined subactions (*RdBegin*, *WriteBegin*, ...). Processes work only on their local copies of the registers ($reg(p_i) \triangleright \mathbf{S}_{\mathfrak{C}}(p_i)$).

A Localview for the α-algorithm comprises p_i's process state, the module for the shared memory, and the identity of p_i. The module for shared memory is adopted identically from configurations to Localviews since the shared memory can be accessed by all processes and, furthermore, unintended access by wrong modeling is avoided by the use of the defined subactions. In a Localview lv, a process can access its process state $\mathbf{S}_{lv}$ and the respective entries $rnd \triangleright \mathbf{S}_{lv}$, $v \triangleright \mathbf{S}_{lv}$, $rtd \triangleright \mathbf{S}_{lv}$, $reg \triangleright \mathbf{S}_{lv}$, and the entries $lre \triangleright reg(p_i) \triangleright \mathbf{S}_{lv}$, $lrww \triangleright reg(p_i) \triangleright \mathbf{S}_{lv}$ and $val \triangleright reg(p_i) \triangleright \mathbf{S}_{lv}$.

Initially, the read-oriented view of every process for every register contains no elements and is, therefore, the emptyset (see Figure 5.26). Every register has an initial value of $(\ 0,\ i,\ \bot\ )$ and is initially in phase **rIdle**. The round number and the value for the first invocation of Alpha for a process p_i are chosen when p_i transits from phase Idle to phase Publishing. Therefore, initially p_i stores an arbitrary round number $r' \in \mathbb{N}$ and an arbitrary value $v' \in \mathbb{I}$ in its process state. The list of registers to read $rtd \triangleright \mathbf{S}_{R(0)}$ is initially the empty list [ ] and the copy of the registers is set to the initial value of the registers.

$$\begin{aligned}
\text{initviewr}(p_i) &\triangleq \emptyset \\
\text{initval}_{al}(p_i) &\triangleq (\ 0,\ i,\ \bot\ ) \\
\text{initsh}(p_i) &\triangleq (\ \mathbf{rIdle},\ \text{initval}_{al}(p_i),\ \text{initviewr}\ )
\end{aligned}$$

$$\text{Init}_{al} \triangleq \{ \begin{pmatrix} (\ r',\ v',\ \text{Idle},\ [\,],\ \text{initval}_{al}\ ) \\ \text{initsh} \end{pmatrix} \mid r' \in \mathbb{N} \wedge v' \in \mathbb{I} \}$$

Figure 5.26.: Initial definitions for the α-algorithm

p_i starts a new invocation by executing the action $\boldsymbol{ProcPublishRoundBegin}^{\downarrow}$ (see Figure 5.27). A new invocation can be started if p_i either has never invoked Alpha and is in phase Idle or has finished the last invocation and is in phase Returned_v for some $v \in (\mathbb{I} \cup \{\ \bot\ \})$.

p_i chooses a new round number (which must be from p_i's set of round numbers and must be greater than the one used before), adopts it to its state, and starts writing this number to the register entry *lre*. Finally, p_i transits to phase Publishing. Both actions $\mathit{ProcPublishRoundBegin}^{\downarrow}$ and $\mathit{ProcPublishRoundEnd}^{\downarrow}$ (see Listing 5.27) represent the write statement in line 4 of the pseudo code. We model the write by using the subactions *WriteBegin* and *WriteEnd*. Being in phase Publishing p_i executes $\boldsymbol{\mathit{ProcPublishRoundEnd}}^{\downarrow}$ to finish the write to p_i's register. In the same step, p_i non-deterministically determines the order in which p_i will read from processes' registers. This is done by choosing a list pL of all processes and storing this list to the local variable $\mathit{rtd} \triangleright \mathbf{S}_{lv}$. In this step, p_i transits to phase Fetching. Actions $\mathit{ProcFetching}^{\downarrow}$

$$\begin{aligned}
\mathit{ProcPublishRoundBegin}^{\downarrow}(lv, lv') \triangleq\ & \exists v' \,.\, \exists r' \in \mathrm{ReqNo}(\mathbf{ID}_{lv}) \,.\, r' > \mathit{rnd} \triangleright \mathbf{S}_{lv} \\
& \wedge \mathit{phs} \triangleright \mathbf{S}_{lv} \in \{\ \mathrm{Idle}, \mathrm{Returned}_v \ \mid\ v \in (\mathbb{I} \times \{\ \bot\ \})\ \} \\
& \wedge \mathbf{S}_{lv'} = (\ r',\ v',\ \mathrm{Publishing},\ \mathit{rtd} \triangleright \mathbf{S}_{lv},\ \mathit{reg} \triangleright \mathbf{S}_{lv}\ ) \\
& \wedge\ \mathit{WriteBegin}(\mathbf{Sh}_{lv}, \mathbf{Sh}_{lv'}, \mathbf{ID}_{lv}, (\ r',\ \text{-},\ \text{-}\ ))
\end{aligned}$$

$$\begin{aligned}
\mathit{ProcPublishRoundEnd}^{\downarrow}(lv, lv') \triangleq\ & \exists pL \,.\, \mathrm{set}(pL) = \mathbb{P} \wedge \mathrm{distinct}(pL) \\
& \wedge \mathit{phs} \triangleright \mathbf{S}_{lv} = \mathrm{Publishing} \\
& \wedge \mathbf{S}_{lv'} = (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ v \triangleright \mathbf{S}_{lv},\ \mathrm{Fetching},\ pL,\ \mathit{reg} \triangleright \mathbf{S}_{lv}\ ) \\
& \wedge\ \mathit{WriteEnd}(\mathbf{Sh}_{lv}, \mathbf{Sh}_{lv'}, \mathbf{ID}_{lv})
\end{aligned}$$

Figure 5.27.: Actions $\mathit{ProcPublishRoundBegin}^{\downarrow}$ and $\mathit{ProcPublishRoundEnd}^{\downarrow}$

and $\mathit{ProcFetchingStep}^{\downarrow}$ (see Figure 5.28) model the read of the registers in lines 6 and 18. For starting the first read cycle (line 6), a process p_i has to be in phase Fetching, for the second cycle, p_i must be in phase FetchingAgain. By executing $\boldsymbol{\mathit{ProcFetching}}^{\downarrow}$ p_i starts reading the first register in list $\mathit{rtd} \triangleright \mathbf{S}_{lv}$ (the register is obtained by the hd operation). If p_i is in phase Fetching, p_i proceeds to phase FetchingWait, else, if p_i is in phase FetchingAgain, it transits to phase FetchingAgainWait. To finish a read operation on a register, p_i performs operation $\boldsymbol{\mathit{ProcFetchingStep}}^{\downarrow}$. This operation stores the read value in p_i's process state by updating the local copy of the respective register. Furthermore, the head of the list is removed from the list to enable the read of the next register. To start the read of the next register, p_i goes back to phase Fetching (respectively FetchingAgain for the second cycle).

If p_i is in phase Fetching and there are no more registers to read, by executing action $\boldsymbol{\mathit{ProcFetchingFinish}}^{\downarrow}$ (see Figure 5.29) p_i chooses the value to write and starts writing its round number and the value to its own register (entries *lrww*, *val*) using the subaction *WriteBegin*. This action is only enabled, if p_i has read no higher round number than its own in the *lre* entries. $\mathit{ProcFetchingFinish}^{\downarrow}$ sets the phase to Proposing. If p_i is in phase Fetching or FetchingAgain, there are no more registers to read and p_i has read a higher round number, then p_i can execute $\boldsymbol{\mathit{Abort}}^{\downarrow}$ to abort the invocation. In this case the phase is set to $\mathrm{Returned}_{\bot}$.

In phase Proposing, p_i processes action $\mathit{ProcPropose}^{\downarrow}$ to finish the writing that was started

$$\text{nextPhs}(p) \triangleq \begin{cases} \text{FetchingWait} & \text{, if } p = \text{Fetching} \\ \text{FetchingAgainWait} & \text{, if } p = \text{FetchingAgain} \\ \text{Fetching} & \text{, if } p = \text{FetchingWait} \\ \text{FetchingAgain} & \text{, if } p = \text{FetchingAgainWait} \end{cases}$$

$$\begin{aligned} \mathit{ProcFetching}^{\downarrow}(lv, lv') \triangleq\ & \mathit{phs} \rhd \mathbf{S}_{lv} \in \{\ \text{Fetching},\ \text{FetchingAgain}\ \} \\ & \wedge\ \mathit{rtd} \rhd \mathbf{S}_{lv} \neq [\ ] \\ & \wedge\ \mathbf{S}_{lv'} = (\ \mathit{rnd} \rhd \mathbf{S}_{lv},\ v \rhd \mathbf{S}_{lv},\ \text{nextPhs}(\mathit{phs} \rhd \mathbf{S}_{lv}),\ \mathit{rtd} \rhd \mathbf{S}_{lv},\ \mathit{reg} \rhd \mathbf{S}_{lv}\ ) \\ & \wedge\ \mathit{RdBegin}(\mathbf{Sh}_{lv}, \mathbf{Sh}_{lv'}, \mathbf{ID}_{lv}, \text{hd}(\mathit{rtd} \rhd \mathbf{S}_{lv})) \end{aligned}$$

$$\begin{aligned} \mathit{ProcFetchingStep}^{\downarrow}(lv, lv') \triangleq\ & \exists \mathit{cnt}\ .\ \mathit{phs} \rhd \mathbf{S}_{lv} \in \{\ \text{FetchingWait},\ \text{FetchingAgainWait}\ \} \\ & \wedge\ \mathbf{S}_{lv'} = (\mathit{rnd} \rhd \mathbf{S}_{lv}, v \rhd \mathbf{S}_{lv}, \text{nextPhs}(\mathit{phs} \rhd \mathbf{S}_{lv}), \text{tl}(\mathit{rtd} \rhd \mathbf{S}_{lv}), \\ & \qquad\qquad \mathit{reg} \rhd \mathbf{S}_{lv}[(\text{hd}(\mathit{rtd} \rhd \mathbf{S}_{lv})) := \mathit{cnt}]) \\ & \wedge\ \mathit{RdEnd}(\mathbf{Sh}_{lv}, \mathbf{Sh}_{lv'}, \mathbf{ID}_{lv}, \text{hd}(\mathit{rtd} \rhd \mathbf{S}_{lv}), \mathit{cnt}) \end{aligned}$$

Figure 5.28.: Actions *ProcFetching*$^{\downarrow}$ and *ProcFetchingStep*$^{\downarrow}$

with *ProcFetchingFinish*$^{\downarrow}$. It furthermore chooses the list of processes for the next read cycle and proceeds to phase FetchingAgain.

After registers have been read for the second time (using process-actions*ProcFetching*$^{\downarrow}$ and *ProcFetchingStep*$^{\downarrow}$), p_i is again in phase FetchingAgain. If there are no more registers to read, p_i can finally return a value if it has not read a *lre* value higher than its own round number. By executing ***ProcReturn***$^{\downarrow}$ p_i sets the phase to Returned$_v$ where v is the value p_i returns, which is the value p_i has written to its register. If p_i has read a higher value, p_i executes *Abort*$^{\downarrow}$ to abort the invocation.

If not already crashed, a process might always crash by executing ***ProcCrash***$^{\downarrow}$. In this case the phase is set to Crashed.

Figure 5.30 shows the final definitions for the formal α-algorithm. Functions C2LV$_{al}$ and LV2C$_{al}$ are used to convert configurations to Localviews and vice versa. The function C2LV$_{al}$ generates a Localview out of a configuration $\mathfrak{C}$ for a process p_i by returning a vector of the

- state of a process: $\mathbf{S}_{\mathfrak{C}}(p_i)$
- the state of the registers: $\mathbf{Sh}_{\mathfrak{C}}$
- the idenity of p_i

LV2C$_{al}$ embeds a Localview lv back in to a given configuration $\mathfrak{C}$ by

- setting the state of process p_i to the process state $\mathbf{S}_{lv}$ of the Localview
- adopting the state of the registers back into the configuration

$$\text{valtowrite}(v, v') \triangleq \begin{cases} v' & \text{, if } v = \bot \\ v & \text{, else} \end{cases}$$

$$\begin{aligned}
\mathit{ProcFetchingFinish}^{\downarrow}(lv, lv') \triangleq\ & \mathit{phs} \triangleright \mathbf{S}_{lv} = \text{Fetching} \\
& \wedge \neg\, (\exists p_j \in \mathbb{P} \,.\, \mathit{lre} \triangleright \mathit{reg}(p_j) \triangleright \mathbf{S}_{lv} > \mathit{rnd} \triangleright \mathbf{S}_{lv}) \\
& \wedge\ \mathit{rtd} \triangleright \mathbf{S}_{lv} = [\,] \\
& \wedge\ (\exists p_j \in \mathbb{P} \,.\, \forall p_k \in \mathbb{P} \,.\, \mathit{lrww} \triangleright \mathit{reg}(p_j) \triangleright \mathbf{S}_{lv} \geq \mathit{lrww} \triangleright \mathit{reg}(p_k) \triangleright \mathbf{S}_{lv} \\
& \wedge\ \mathbf{S}_{lv'} = (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ v \triangleright \mathbf{S}_{lv},\ \text{Proposing},\ \mathit{rtd} \triangleright \mathbf{S}_{lv},\ \mathit{reg} \triangleright \mathbf{S}_{lv}\ ) \\
& \wedge\ \mathit{WriteBegin}(\mathbf{Sh}_{lv}, \mathbf{Sh}_{lv'}, \mathbf{ID}_{lv}, \\
& \qquad\qquad (\ \text{-},\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ \text{valtowrite}(\mathit{val} \triangleright \text{Reg}(p_j) \triangleright \mathbf{S}_{lv},\ v \triangleright \mathbf{S}_{lv})\ ))
\end{aligned}$$

$$\begin{aligned}
\mathit{Abort}^{\downarrow}(lv, lv') \triangleq\ & \mathit{phs} \triangleright \mathbf{S}_{lv} \in \{\ \text{Fetching},\ \text{FetchingAgain}\ \} \\
& \wedge\ \mathit{rtd} \triangleright \mathbf{S}_{lv} = [\,] \\
& \wedge\ (\exists p_j \in \mathbb{P} \,.\, \mathit{lre} \triangleright \mathit{reg}(p_j) \triangleright \mathbf{S}_{lv} > \mathit{rnd} \triangleright \mathbf{S}_{lv}) \\
& \wedge\ \mathbf{S}_{lv'} = (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ v \triangleright \mathbf{S}_{lv},\ \text{Returned}_{\bot},\ \mathit{rtd} \triangleright \mathbf{S}_{lv},\ \mathit{reg} \triangleright \mathbf{S}_{lv}\ ) \\
& \wedge\ \mathbf{Sh}_{lv'} = \mathbf{Sh}_{lv}
\end{aligned}$$

$$\begin{aligned}
\mathit{ProcPropose}^{\downarrow}(lv, lv') \triangleq\ & \exists pL \,.\, \text{set}(pL) = \mathbb{P} \wedge \text{distinct}(pL) \\
& \wedge\ \mathit{phs} \triangleright \mathbf{S}_{lv} = \text{Proposing} \\
& \wedge\ \mathbf{S}_{lv'} = (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ v \triangleright \mathbf{S}_{lv},\ \text{FetchingAgain},\ pL,\ \mathit{reg} \triangleright \mathbf{S}_{lv}\ ) \\
& \wedge\ \mathit{WriteEnd}(\mathbf{Sh}_{lv}, \mathbf{Sh}_{lv'}, \mathbf{ID}_{lv})
\end{aligned}$$

$$\begin{aligned}
\mathit{result}(\mathbf{Sh}, p_i) \triangleq\ & \triangleright\, \text{view}^{\text{w}} \triangleright \mathbf{Sh}(p_i) \\
\mathit{ProcReturn}^{\downarrow}(lv, lv') \triangleq\ & \mathit{phs} \triangleright \mathbf{S}_{lv} = \text{FetchingAgain} \\
& \wedge\ \mathit{rtd} \triangleright \mathbf{S}_{lv} = [\,] \\
& \wedge \neg\, (\exists p_j \in \mathbb{P} \,.\, \mathit{lre} \triangleright \mathit{reg}(p_j) \triangleright \mathbf{S}_{lv} > \mathit{rnd} \triangleright \mathbf{S}_{lv}) \\
& \wedge\ \mathbf{S}_{lv'} = (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ v \triangleright \mathbf{S}_{lv},\ \text{Returned}_{\mathit{result}(\mathbf{Sh}_{lv}, \mathbf{ID}_{lv})},\ \mathit{rtd} \triangleright \mathbf{S}_{lv},\ \mathit{reg} \triangleright \mathbf{S}_{lv}\ ) \\
& \wedge\ \mathbf{Sh}_{lv'} = \mathbf{Sh}_{lv}
\end{aligned}$$

$$\begin{aligned}
\mathit{ProcCrash}^{\downarrow}(lv, lv') \triangleq\ & \mathit{phs} \triangleright \mathbf{S}_{lv} \neq \text{Crashed} \\
& \wedge\ \mathbf{S}_{lv'} = (\ \mathit{rnd} \triangleright \mathbf{S}_{lv},\ v \triangleright \mathbf{S}_{lv},\ \text{Crashed},\ \mathit{rtd} \triangleright \mathbf{S}_{lv},\ \mathit{reg} \triangleright \mathbf{S}_{lv}\ ) \\
& \wedge\ \mathbf{Sh}_{lv'} = \mathbf{Sh}_{lv}
\end{aligned}$$

Figure 5.29.: Actions *ProcFetchingFinish*$^{\downarrow}$, *Abort*$^{\downarrow}$, *ProcReturn*$^{\downarrow}$, *ProcPropose*$^{\downarrow}$ and *ProcCrash*$^{\downarrow}$

We define the set of process-actions $\mathbf{\Phi_{al}}$ as the set that contains the lifted versions of the non-lifted actions defined before. The α-algorithm algorithm uses no events, therefore the set Ψ_{al} is the emptyset.

In the pseudo code, it seems very obvious that a process p_i successively executes one line of code after the other. In a formal model, it becomes clear that nothing works without a notion of fairness. To exclude runs where only some processes make progress, we need weak fairness for our actions. Hence, we assume that every action always is either eventually disabled or executed. We require this assumption for all defined actions but $\mathit{ProcCrash}^{\downarrow}$ because assuming weak fairness for $\mathit{ProcCrash}^{\downarrow}$ would require every process to crash eventually.

We conclude with the tuple $(\ \mathrm{Init}_{al},\ \Phi_{al},\ \Psi_{al},\ \{\ \mathit{Fairness}\ \},\ \mathrm{C2LV}_{al},\ \mathrm{LV2C}_{al}\ )$ that defines the formal Distributed Algorithm $\mathfrak{A}_{al}$.

$$
\begin{aligned}
\mathrm{C2LV}_{al}(\mathfrak{C}, p_i) &\triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}(p_i), \\ \mathbf{Sh}_{\mathfrak{C}} \\ p_i \end{pmatrix} \\
\mathrm{LV2C}_{al}(lv, p_i, \mathfrak{C}) &\triangleq \begin{pmatrix} \mathbf{S}_{\mathfrak{C}}[p_i := \mathbf{S}_{lv}] \\ \mathbf{Sh}_{lv} \end{pmatrix} \\
\mathrm{Correct}(R) &\triangleq \{\ p_i \in \mathbb{P}\ \mid\ \forall t \in \mathbb{T}\ .\ \mathit{phs} \triangleright \mathbf{S}_{R(t)} \neq \mathrm{Crashed}\ \} \\
\mathit{WeakFairness}(R, \mathcal{A}) &\triangleq \forall t \in \mathbb{T}\ .\ \forall p_i \in \mathrm{Correct}(R)\ .\ \exists t' \geq t\ . \\
&\qquad\qquad (R(t') \rightarrow_{p_i:\mathcal{A}} R(t'+1)) \vee \neg\, \mathbf{Enabled}_A(\mathcal{A}, R(t'), p_i) \\
\mathit{FairActions} &\triangleq \Big\{\ \mathit{ProcPublishRoundEnd}^{\downarrow},\ \mathit{ProcFetching}^{\downarrow},\ \mathit{ProcFetchingStep}^{\downarrow}\ \Big\} \\
&\quad \cup \Big\{\ \mathit{ProcFetchingFinish}^{\downarrow},\ \mathit{ProcReturn}^{\downarrow},\ \mathit{ProcPropose}^{\downarrow},\ \mathit{Abort}^{\downarrow}\ \Big\} \\
\mathit{Fairness}(R) &\triangleq \forall \mathcal{A}^{\downarrow} \in \mathit{FairActions}\ .\ \mathit{WeakFairness}(R, \mathrm{Lift}(\mathcal{A}^{\downarrow})) \\
\Phi_{al} &\triangleq \Big\{\ \mathrm{Lift}(\mathit{ProcPublishRoundBegin}^{\downarrow}),\ \mathrm{Lift}(\mathit{ProcPublishRoundEnd}^{\downarrow})\ \Big\} \\
&\quad \cup \Big\{\ \mathrm{Lift}(\mathit{ProcFetching}^{\downarrow}),\ \mathrm{Lift}(\mathit{ProcFetchingStep}^{\downarrow}),\ \mathrm{Lift}(\mathit{ProcFetchingFinish}^{\downarrow})\ \Big\} \\
&\quad \cup \Big\{\ \mathrm{Lift}(\mathit{Abort}^{\downarrow}),\ \mathrm{Lift}(\mathit{ProcReturn}^{\downarrow}),\ \mathrm{Lift}(\mathit{ProcPropose}^{\downarrow}),\ \mathrm{Lift}(\mathit{ProcCrash}^{\downarrow})\ \Big\} \\
\Psi_{al} &\triangleq \emptyset \\
\mathfrak{A}_{al} &\triangleq (\ \mathrm{Init}_{al},\ \Phi_{al},\ \Psi_{al},\ \{\ \mathit{Fairness}\ \},\ \mathrm{C2LV}_{al},\ \mathrm{LV2C}_{al}\ )
\end{aligned}
$$

Figure 5.30.: Formal α-algorithm

5.4.3. Proof Issues

For the α-algorithm, we restrict ourselves to the safety property: αQuasi-agreement (the property with the most extensive proof) and the liveness property αTermination. As the work of

Guerraoui and Raynal [GR07] shows, the paper proofs for αConditional non-$\bot$ convergence and αValidity are very short such that in a theorem prover they would only require many technical arguments for the preservation of invariants during a run.

For αTermination we have to show that every correct process eventually returns a value (maybe $\bot$) in every invocation. That requires a definition clarifying what it means for a process to be invoked in terms of our model.

Hence, we define a predicate invoked:

Definition 5.4.3.1 (Invoked)
Let p_i be a process, $\mathfrak{C} \in \mathbb{C}$ a configuration, $r \in \mathbb{N}$ be a round number and v be a (input) value. We define:

$$\text{invoked}_{\mathfrak{C}}(p_i, r, v) \triangleq \mathit{phs} \triangleright \boldsymbol{S}_{\mathfrak{C}}(p_i) = \text{Publishing} \wedge \mathit{rnd} \triangleright \boldsymbol{S}_{\mathfrak{C}}(p_i) = r \wedge v \triangleright \boldsymbol{S}_{\mathfrak{C}}(p_i) = v$$

For αTermination we show that every correct process p_i that has been invoked with round number r and value v at time t eventually is in phase Returned_x for some $x \in \mathbb{I} \cup \{ \bot \}$. The formal definition for αTermination for our model is:

$$\begin{aligned}\alpha\text{Termination}(R) \triangleq & \forall p_i \in \text{Correct}(R) \,.\, \forall t \in \mathbb{T} \,.\, \forall r \in \mathbb{N} \,.\, \forall v \in \mathbb{I} \,.\, \text{invoked}_{R(t)}(p_i, r, v) \\ & \Rightarrow \left(\exists t' > t \,.\, \exists x \,.\, \mathit{phs} \triangleright \mathbf{S}_{R(t)}(p_i) = \text{Returned}_x\right)\end{aligned}$$

The only argument for αTermination by Guerraoui and Raynal is ‘As it is wait-free, the algorithm described ... trivially satisfies the termination property’. Since it is hard to find a formal definition for *wait-free* in terms of our model, our proof for Termination is based on the assumption of weak-fairness (see Section 5.4.2). This assumption enables us to show that

- After a correct process p_i invoked Alpha, p_i will eventually execute action $\mathit{ProcPublishRoundEnd}^{\downarrow}$ ($\hookrightarrow$ Isabelle[18]).
- After a correct process p_i has visited phase Fetching, p_i's list of registers to read will eventually be empty ($\hookrightarrow$ Isabelle[19]).
- If a correct process p_i is in phase Fetching or FetchingAgain in some configuration $\mathfrak{C}$ and it has a non-empty list of registers to read, then p_i will eventually execute $\mathit{ProcFetching}^{\downarrow}$ in some configuration $\mathfrak{C}'$ with $\mathbf{S}_{\mathfrak{C}} = \mathbf{S}_{\mathfrak{C}'}$ ($\hookrightarrow$ Isabelle[20]).
- After a correct process p_i has visited phase FetchingAgain and has an empty list of registers to read, p_i eventually executes $\mathit{ProcReturn}^{\downarrow}$ ($\hookrightarrow$ Isabelle[21]).
- If a correct process p_i is in phase FetchingWait or FetchingAgainWait in some configuration $\mathfrak{C}$, then p_i will eventually execute $\mathit{ProcFetchingStep}^{\downarrow}$ in some configuration $\mathfrak{C}'$ with $\mathbf{S}_{\mathfrak{C}} = \mathbf{S}_{\mathfrak{C}'}$ ($\hookrightarrow$ Isabelle[22]).

[18]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `InvokedLeadsToPPRE`
[19]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `FetchingLeadsToPFF`
[20]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `FetchingLeadsToPF`
[21]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `FetchingAgainLeadsToPR`
[22]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `FetchingWaitLeadsToPFS`

- If a correct process p_i is in phase Fetching or FetchingAgain in some configuration $\mathfrak{C}$ and it has a non-empty list of registers to read, then p_i will eventually be in the same phase in some configuration $\mathfrak{C}'$ and p_i's registers to read equal the tail of p_i's registers to read in $\mathfrak{C}$ ($\hookrightarrow$ Isabelle[23]).
- If a correct process p_i is in phase Proposing in some configuration $\mathfrak{C}$, then p_i will eventually execute action $\mathit{ProcPropose}^{\downarrow}$ ($\hookrightarrow$ Isabelle[24]).
- After a correct process p_i invoked Alpha, p_i will eventually execute action $\mathit{ProcReturn}^{\downarrow}$ or return the $\bot$ value (by executing $\mathit{Abort}^{\downarrow}$) ($\hookrightarrow$ Isabelle[25]).

Finally, it is easy to show that the last item implies αTermination. Note that again there is an abundance of detail required for the formal proof compared to the proof of Guerraoui and Raynal (see above). At this point we emphasize again that for verifying correctness, the pseudo code Guerraoui and Raynal use for their reasoning is obviously an inedaquate representation of the algorithm since it hides important details that are relevant for the proofs: single lines of pseudo code (lines 6 and 18) are obviously loops, although they look like single statements. Of course, loops are relevant for every proof of Termination but they are not even mentioned in descriptions and proofs in [GR07].

To show the α-algorithm exhibits the αQuasi-agreement property, we have to prove that whenever two invocations return values v_i and v_j and $v_i \neq \bot \neq v_j$, then v_i equals v_j. Formally we define:

$$\begin{aligned}\alpha\text{Quasi-agreement}(R) \triangleq\, & \forall p_i, p_j \in \mathbb{P} \,.\, \forall t_i, t_j \in \mathbb{T} \,.\, \forall v_i, v_j \,.\, v_i \neq \bot \wedge v_j \neq \bot \\ & \wedge\, \mathit{phs} \triangleright \mathbf{S}_{R(t_i)}(p_i) = \text{Returned}_{v_i} \wedge \mathit{phs} \triangleright \mathbf{S}_{R(t_j)}(p_j) = \text{Returned}_{v_j} \\ & \Rightarrow v_i = v_j\end{aligned}$$

We adopted the structure of the proof for αQuasi-agreement from Guerraoui and Raynal [GR07]. In the following, we will give a sketch of the proof and point out some details we consider of importance in comparison with the given paper proof.

Agreement is shown by the proof that every returned value v' of an invocation I' by a process p_j equals the value that is returned by the invocation I (invoked by a process p_i) with the smallest round number r that ever returned a value $v \neq \bot$ (the formal proof for a respective lemma is given in $\hookrightarrow$ Isabelle[26]). So, analogously to [GR07] let us consider two invocations I (the one with the smallest round number) and I' that returned values $v \neq \bot$ and $v' \neq \bot$. We use the following time instant definitions for these two invocations (cf. [GR07]):

- t_{propE} is the time at which I finishes the write of entries *lrww* and *val* by executing $\mathit{ProcPropose}^{\downarrow}$.
- t_{faS} is the time at which p_i in I (in phase FetchingAgain) starts reading the register Reg_{p_j} for the second time using action $\mathit{ProcFetching}^{\downarrow}$.

[23]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `FetchingLeadsToFetching`
[24]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `ProposingLeadsToPP`
[25]Isabelle/HOL theory: `AlphaVerification.thy`, Lemma(s): `InvokedLeadsToProcReturn`
[26]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `AgreementToSmallest`

- t_{pubE} is the time at which p_j in I' finishes the write of entry *lre* by executing action *ProcPublishRoundEnd*$^{\downarrow}$.

- t_{fS} is the time at which p_j in I' (in phase Fetching) starts reading Reg_{p_i} for the first time using action *ProcFetching*$^{\downarrow}$.

It is easy to see that $t_{propE} < t_{faS}$ and $t_{pubE} < t_{fS}$. Let r' be the round number p_j uses in invocation I'. If p_j finished writing round number r' to entry *lre* before p_i started reading Reg_{p_j}, then p_i would read a round number higher than its own round number (recall that r is the smallest round number for which a value is returned) and would have aborted the invocation. Therefore, we have $t_{faS} < t_{pubE}$ and hence $t_{propE} < t_{faS} < t_{pubE} < t_{fS}$. Thus, when p_j reads register Reg_{p_i} at time t_{fS}, p_j must read a value with a *lrww* entry $x \geq r$ (because processes use increasing round numbers). Hence, the value p_j chooses as a return value must have an *lrww* entry greater or equal to x. Let us assume this value $y \geq x$ has been written by a process p_k in invocation I''. As marked by Guerraoui and Raynal, the case where $x = y$ is trivial: since processes use unique round numbers this implies p_j adopted the value from p_i and $I = I''$ and hence, we have $v = v'$. For the case $x \neq y$ Guerraoui and Raynal argue that if $I \neq I''$ the pair of invocations I, I'' can be considered as we considered I, I' previously. As a result, they claim that finally this yields an invocation that adopted the value from I. This, of course, is much too sloppy to be used as a formal proof in a theorem prover. Formally, this idea yields an inductive proof over the round number of r'. Therefore, the proof in $\hookrightarrow$ Isabelle[27] is done by induction on the round number of I'.

Much effort was required to formally do this proof. The main concern was that there are many passages where Guerraoui and Raynal reason over the contents of registers and argue that if a process had read some register value x it would have aborted the invocation, but actually processes decisions are not taken on the base of register values but only on the base of the copies of these values. Therefore formal reasoning must be done with respect to the values of the copies and, hence, must always consider that this value of the copy must have been read by the process from the respective register and that the value in the register must initially have been written by some respective process. Moreover, as already described, Guerraoui and Raynal's arguments are only based on the event-based definitions for Regular Registers given by Lamport. Therefore, we had to transfer every proposition they made to the model introduced in Section 2.3.7. These two concerns and the fact that the scheme for the induction was not entirely clear from the beginning made reasoning for this algorithm extremely difficult.

Furthermore, some errors in the work of Guerraoui and Raynal [GR07] impeded our work: In a remark they claim that the test in line 8 (see pseudo code) would not be necessary. As a consequence we omitted this test until we recognized, that the justification of Guerraoui and Raynal that this test is never used in the proof, is wrong. Actually, it is needed, but Guerraoui and Raynal erronously refer to some line number greater than their greatest line number (see [GR07]: although their original pseudo code comprises only 9 lines they write '...does not return $\perp$ at line 12 ...'). We do not claim that the algorithm does not satisfy αQuasi-agreement if this test is omitted, but in our point of view the proof as it is written down by Guerraoui and

[27]Isabelle/HOL theory: `RotatingCoordVerification.thy`, Lemma(s): `AgreementToSmallest`

Raynal, crucially requires this test. As our intention was not to invent a new proof but only to redo the proof in a theorem proving environment, our solution here was to insert the test back into the algorithm allowing us to finish the proof for αQuasi-agreement.

The previous sections (see Sections 5.1.2, 5.2.2 and 5.3.2) showed how our model performs for algorithms using message passing. This new example shows how we finally succeeded in applying our model on an algorithm that uses shared memory for communication.

6. Conclusion

Although proofs for fault tolerant algorithms are considered an integral part of every published work in the field of Distributed Algorithms, pseudo code is a widespread representation for such algorithms. In our point of view, though, formal correctness proof should be based on formal semantics. Since pseudo code lacks formal semantics, proofs derived on the basis of pseudo code are informal and mostly incomplete. Nevertheless, pseudo code representation does have some advantages: in that it is usually easily readable and understood without previous (formal) knowledge. Furthermore, in some way it resembles implementation thus allowing programmers to use that adhoc intuition in implementing an algorithm given by a pseudo code. Based on the high level description, a reader is able to sort out the fundamental aspects of an algorithm without having to learn a new model and without any knowledge of formal methods. Therefore, the essential ideas of an algorithm can be recognized quickly through the pseudo code representation. In consensus with these arguments, our representations of the algorithms in Chapter 5 also include a pseudo code representation. Nevertheless, in our opinion, verification of an algorithm is only possible using a formal model so that we have analysed existing approaches to model distributed algorithms succeeding in developing a new formal model that is similar to some well-known approaches such as [abstract] state machines and the TLA-format.

Reasoning based upon a formal model requires a higher effort that has to be spent for doing the proofs. But we think that this process of formally writing down all details helps to identify what is really meant by the properties, assumptions, and mechanisms used to describe an algorithm. For example scientific literature (e.g. [FMN07], [CT96], [Lyn96]) lists only three properties for Distributed Consensus: Validity, Agreement and Termination. Although nearly every work on Consensus agrees that decision for a value must be irrevocable, only in the work of Charron-Bost (e.g. [CBM09]) Irrevocability is an explicit property. Another example can be found in different definitions of a *broadcast*. In some works as e.g. [Fuz08] the term *broadcast* implies that the underlying infrastructure differs from point-to-point channels. But the work of Francalanza and Hennessy [FH07] obviously uses the term broadcast synonymously to sending a message to all processes. All in all, we agree with Aguilera [Agu10] that it should be common practice to analyse and explain results intuitively and formally to profit from the benefit of both approaches.

6.1. Contributions of our Modeling Approach

The main contribution of our model is that all parts are formal enough to be used within a theorem prover. Indeed we formalized the model within the interactive theorem proving environment Isabelle/HOL. In contrast to other formal approaches to formal modeling, as e.g. process calculi, our model is based on higher order logic, and, therefore, readable at hand for anyone who has some basic knowledge of logics. Our introduced TLA-like actions can be seen as a refinement of the pseudo code, i.e., a method to enrich the pseudo code with formal semantics. Like many common approaches, the model uses interleaving semantics based on atomic steps executed by processes and events. Starting from defined initial states, the formal definition of actions and events enable us to compute the set of runs of the algorithm and, therefore, to formally prove that the algorithm never violates the specified safety properties. Moreover, we provide mechanisms to integrate fairness assumptions into the model that enable deduction of liveness properties, which may rely on additional assumptions.

In Section 2.4 we explained how safety and liveness properties are specified in terms of our model and gave explicit definitions for those classes of properties in terms of our model. Finally, we illustrated our terms and definitions by the example of the properties of Distributed Consensus.

In contrast to other approaches, our model can be applied to a larger group of algorithms and is not limited to round based algorithms (as e.g. the HO-model [CBS07]) nor to algorithms that use only message passing for communication. The introduced mechanisms for communication (message passing, broadcasts and shared memory) support a large spectrum of possible applications. It is, moreover, possible to easily adapt and extend these mechanisms if new applications require a different communication infrastructure. As described in Sections 2.3.5, 2.3.6, and 2.3.7 the provided approaches for message passing, broadcasts, and shared memory comprise:

- a model for Quasi-reliable point-to-point communication without message loss
- a model for Quasi-reliable point-to-point communication with message loss
- a model for Reliable Broadcasts
- a model for Regular Registers

To the best of our knowledge, the formal model for shared memory is the first state based model of Regular Registers that has been formally defined in a theorem proving environment. As described in Section 2.3.7 we also provided a proof that our model is equivalent to the original definitions given by Leslie Lamport [Lam85].

A crucial characteristic for different fault tolerant algorithms is the respective failure model the algorithms assume. We believe that every possible failure model can be integrated in our model and have provided examples for the Crash-Failure and for the Fail-Recovery model (see Section 2.3.8 and Chapter 5).

As described in Section 2.3.2, our model supports different methods to guarantee consistency with respect to the notion of distribution. The new concept of *Localviews* and *Deterministic non-lifted Actions*(DnAs) (see Section 2.3.2) limits the data a process is able to access locally. Thereby, the limit is defined by the designer of the model. Our concept exhibits advantages in design as well as for the verification of the algorithm: in contrast to [Fuz08], process-actions are defined by describing only local changes and the application of the introduced DnAs also works well with automated proof strategies (e.g. the *auto* method in Isabelle/HOL). Furthermore, by the introduced constraints that have to be satisfied for our definition of a Distributed Algorithm (see Section 2.3.2), we ensure that processes only read and manipulate data that they can access locally.

The presented case studies demonstrated how our introduced model can be used to mechanically verify distributed algorithms.

6.2. Contributions of our Verification Strategies

Mostlikely, the most important observation regarding our proof strategies is that many ideas from the existing paper proofs could be transferred to the proofs we carried out in the theorem prover. This implies that basically a formal proof uses the same arguments as informal argumentations but many details that often are omitted in informal proofs have to be particularised.

For application of a semi-automated theorem prover, we experienced that much effort (compared to the paper proofs) will be spent in considering cases that seem to be too 'trivial' to be mentioned in a paper proof. In Chapter 3, we presented different strategies to reduce the work for such trivial cases. Furthermore, we analyzed the applicability of different proof methods relevant to different classes of properties. We identified the induction on runs as a standard strategy for establishing invariants, and, therefore, as the preferred method to prove safety properties like Validity and Agreement. For more sophisticated properties like Irrevocability, which in contrast to Validity and Agreement, is not a *configuration invariant* (see Section 2.4), we provide a theorem reducing the proof of the property to a simple inductional argument similar to the induction mentioned before.

In Section 3.3 we pointed out that a further helpful method to make assertions about what happened in a run before a certain configuration, is to use history variables. Our introduced modules for Message Histories and Regular Registers and Broadcast Histories support this method.

For the proof of liveness properties, we found that the right strategy must usually depend on the given fairness assumptions. We exemplified this in our proof of Termination. Our different case studies actually showed how proofs for Termination based on different fairness assumptions are carried out.

6.3. Contributions of our Work with Isabelle/HOL

Our work with Isabelle/HOL contributed both to establishing a library for distributed algorithms (see Chapter 4) as also to the verification of different algorithms, demonstrated in our case studies (see Chapter 5).

The library supports the modeling and verification of new algorithms with the theorem proving environment Isabelle/HOL. Fundamental to our formalization is our Isabelle/HOL locale `DistributedAlgorithm`, which is the Isabelle/HOL counterpart of the introduced definition for a Distributed Algorithm in Section 2.3.2. For interpretations of this locale, we provide theorems to support the verification strategies introduced in Chapter 3 and, furthermore, some simple lemmas for deducing basic properties of the algorithms.

The library comprises datatypes and definitions that correspond to the terms defined in Section 2.3 (cf. Sections 4.1 and 4.2). This includes a hierarchy of algorithms (Safe Algorithms, Algorithms, Distributed Algorithms), which is implemented by extending the local `Algorithm`. Furthermore, the concept of *component preserving* actions and events and the corresponding lemmas and theorems are provided to support the verification techniques introduced in Chapter

3. Finally, all modules for communication (as introduced in Sections 2.3.5, 2.3.6 and 2.3.7) are given as Isabelle/HOL theories. This comprises point-to-point message passing communication and shared memory communication. We have provided two versions of point-to-point message passing including one with a failure model, where messages might be lost and one without message loss. The theory for shared memory includes a full formal proof for the equivalence of our state based model for Regular Registers to the event based definitions of Lamport. Well-regulated access and control is guaranteed by the provided subactions for each of the different communication mechanisms. The subactions are also part of the Isabelle/HOL theories. Moreover, the theories provide many basic but important lemmas, which are useful for reasoning over communication.

Case studies chosen are drawn from a common fundamental problem of asynchronous distributed computing: Consensus. We verified four algorithms of different complexity using different failure models and different communication mechanisms. An overview may be found in Figure 6.1. First, we analysed the comparatively simple *Rotating Coordinator* algorithm. It can be represented by a pseudo code with only 6 lines of code (LOC). To verify all four properties of Distributed Consensus, though, we had to write about 3000 lines of proof code in Isabelle/HOL (exclusively the code of the library).

Verification of the more complex examples $\diamond\mathcal{S}$-algorithm and Paxos required more than 8000 lines of proof code respectively, although we only verified safety properties for Paxos. But if we take the lines of pseudo code as a measure, the effort for proving the properties has only grown in linear scale. Furthermore, the shared memory problem considered in the last case study took even more lines of code to verify only two properties. This is both due to the definition and the model of Regular Registers, which is much more complex compared to the more simple model for message passing used before. Our work with Isabelle/HOL also enabled the detection of several errors in existing proofs for the same algorithm as well as some problems with the existing models (see e.g. Section 5.3.3 and Section 2.3.2).

We analyzed the models we designed and the proofs we made and found that a lot of theory can be reused although algorithms and even the communication infrastructure the algorithms use are different. The main ideas are reflected by the theory described in Chapter 2, by the locale definitions and theorems (presented in Chapter 4) and by the ideas for verification strategies described in Chapter 3.

Beside establishing a library and the verification of the algorithms, our work with the theorem prover Isabelle/HOL included the verfication of the framework developed by Guerraoui and Raynal [GR07] for the *Alpha* abstraction (cf. Section 5.4). Furthermore, we expanded upon some basic work on definitions and reasoning with safety and liveness properties (cf. Section 2.4).

In work on the *Alpha* framework, we could establish a formal proof that the algorithm given by Guerraoui and Raynal solves Distributed Consensus if the used *Alpha* satisfies the properties αValidity, αConditional non-$\perp$ convergence, αQuasi-agreement, and αTermination.

Algorithm	**Verified Properties**	**Setting/ Failure model**	**LOC pseudo code**	**LOC proof code**	**Source**
Rotating Coordinator	Validity Agreement Irrevocability Termination	Message passing w/o message loss, Crash-Failure	6	3k	[Tel01]
$\Diamond\mathcal{S}$	Validity Agreement Irrevocability Termination	Quasi-reliable message passing, Reliable Broadcasts, Crash-Failure	30	10k	[CT96]
Paxos	Validity Agreement Irrevocability	Message passing w/ message loss, Fail-Recovery	30	8k	[Lam98]
Alpha for shared memory	αQuasi-agreement αTermination	Regular Registers, Crash-Failure	10	8k	[GR07]

Figure 6.1.: Verified Algorithms (Case Studies)

6.4. Summary and Future Work

In summary, we have developed a framework to formally model Distributed Algorithms, which supports verification in a theorem proving environment. Special focus is placed on communication mechanisms and the notion of distribution. We conclusively showed the applicability of the framework in the verification of four different algorithms using Isabelle/HOL. We additionally worked out results for meta theory on the Alpha-framework and safety and liveness properties.

In the future it would be interesting to develop more standard proof techniques for distributed algorithms and especially for the verification of liveness properties. Lamport's proof rules in [Lam94] could be of use to estabish a formal approach for standard methods to utilize fairness assumptions. Furthermore, it would be interesting to examine if Lamport's notion of machine closure could be integrated into our framework for a better support of Lamport's refinement proofs.

Finally, more libraries for communication mechanisms should be developed, as e.g. additional libraries for shared memory abstractions like Atomic and Safe Registers (concepts are proposed in Chapter 2.3.7). Imaginable is also a library for remote procedure calls as a communication mechanism.

We would of course also be interested in the verification of more and different algorithms in other case studies with our framework. This work has focused on Agreement problems with a special emphasis on Distributed Consensus. A verification of algorithms that solves a different problem with different safety and liveness properties could help in identifying new aspects in modeling and reasoning.

A. Additional Propositions for Message Passing

Quasi-reliable Message Passing

Proposition A.0.0.2 (Send **Properties**)
If there is a Send *step (formally:* $\text{Send}(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) for two Message Histories* $\boldsymbol{Q}$ *and* $\boldsymbol{Q'}$ *and a message* m*, then*

1. $\text{outMsgs}_{\boldsymbol{Q'}} = \text{outMsgs}_{\boldsymbol{Q}} \cup \{ m \}$
2. $\text{outMsgs}_{\boldsymbol{Q}} \subseteq \text{outMsgs}_{\boldsymbol{Q'}}$
3. $\text{transitmsgs}_{\boldsymbol{Q'}} = \text{transitmsgs}_{\boldsymbol{Q}}$
4. $\text{recmsgs}_{\boldsymbol{Q'}} = \text{recmsgs}_{\boldsymbol{Q}}$
5. $\text{msgs}_{\boldsymbol{Q}} \subseteq \text{msgs}_{\boldsymbol{Q'}}$
6. $\text{msgs}_{\boldsymbol{Q'}} = \text{msgs}_{\boldsymbol{Q}} \cup \{ m \}$

Proof. $\hookrightarrow$ Isabelle[1]

Proposition A.0.0.3 (Transmit **Properties**)
If there is a Transmit *step (formally:* $\text{Transmit}(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) for two Message Histories* $\boldsymbol{Q}$ *and* $\boldsymbol{Q'}$ *and a message* m*, then*

1. $\text{outMsgs}_{\boldsymbol{Q}} = \text{outMsgs}_{\boldsymbol{Q'}} \cup \{ m \}$
2. $\text{outMsgs}_{\boldsymbol{Q'}} \subseteq \text{outMsgs}_{\boldsymbol{Q}}$
3. $\text{transitmsgs}_{\boldsymbol{Q'}} = \text{transitmsgs}_{\boldsymbol{Q}} \cup \{ m \}$
4. $\text{recmsgs}_{\boldsymbol{Q'}} = \text{recmsgs}_{\boldsymbol{Q}}$
5. $\text{msgs}_{\boldsymbol{Q}} = \text{msgs}_{\boldsymbol{Q'}}$

Proof. $\hookrightarrow$ Isabelle[2]

Proposition A.0.0.4 (Receive **Properties**)
If there is a Receive *step (formally:* $\text{Receive}(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) for two Message Histories* $\boldsymbol{Q}$ *and* $\boldsymbol{Q'}$ *and a message* m*, then*

[1]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `SendProps,SendMsgChange`
[2]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `TransmitProps`

1. outMsgs $_{\boldsymbol{Q}}$ = outMsgs $_{\boldsymbol{Q'}} \cup \{\ m\ \}$
2. transitmsgs $_{\boldsymbol{Q'}} \subseteq$ transitmsgs $_{\boldsymbol{Q}}$
3. transitmsgs $_{\boldsymbol{Q}}$ = transitmsgs $_{\boldsymbol{Q'}} \cup \{\ m\ \}$
4. recmsgs $_{\boldsymbol{Q'}}$ = recmsgs $_{\boldsymbol{Q}} \cup \{\ m\ \}$
5. msgs $_{\boldsymbol{Q}}$ = msgs $_{\boldsymbol{Q'}}$

Proof.$\hookrightarrow$ Isabelle[3]

Proposition A.0.0.5 (Duplicate **Properties)**
If there is a Duplicate *step (formally:* Duplicate$(\boldsymbol{Q}, \boldsymbol{Q'}, m)$*) for two Message Histories* $\boldsymbol{Q}$ *and* $\boldsymbol{Q'}$ *and a message* m*, then*

1. outMsgs $_{\boldsymbol{Q'}}$ = outMsgs $_{\boldsymbol{Q}}$
2. transitmsgs $_{\boldsymbol{Q'}}$ = transitmsgs $_{\boldsymbol{Q}}$
3. recmsgs $_{\boldsymbol{Q'}}$ = recmsgs $_{\boldsymbol{Q}}$
4. msgs $_{\boldsymbol{Q'}}$ = msgs $_{\boldsymbol{Q}}$

Proof.$\hookrightarrow$ Isabelle[4]

[3]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `ReceiveProps`
[4]Isabelle/HOL theory: `MsgPassNoLoss.thy`, Lemma(s): `DuplicateProps`

Glossary

ProperMemStep
A step which is in accordance with the defined subactions for Regular Registers (cf. Definition 2.3.7.1). 56, 57, 113–116, 118, 121

Admissible Runs
For an Algorithms Admissible Runs are the runs where the defined fairness predicates hold (cf. Definition 2.3.1.9). 24, 68, 83, 155, 205

Agreement
Property of Distributed Consensus: No two processes decide on different values. 5, 6, 26, 68, 69, 125, 153, 193, 198

algorithm
Procedure for calculation which is defined by single steps (cf. Definition 2.3.1.8). Our definition of an algorithm depends on the definition of a Safe Algorithm also comprises fairness assumptions. iii, 1–6, 11–13, 15–27, 35–39, 41, 43, 47, 49, 62–65, 67, 68, 71, 73, 75–79, 82, 83, 85, 127–132, 136–138, 140, 154–158, 170, 172, 174, 175, 177, 186–193, 197, 199

Byzantine
failure model where fa ulty processors can behave arbitrarily. 31, 64

configuration
Global state of the system which is a combination of all the modules (e.g. communication modules or failure detectors) that are present in the system and are used by the algorithm in addition to the state of the involved processes (cf. Definition 2.3.1.1). 20–26, 30–34, 36, 38, 41, 42, 47, 49, 51, 52, 57, 62, 63, 66–73, 75–80, 82, 83, 125, 126, 131, 136, 153, 170, 174, 185, 186, 191, 199, 200, 213

Configuration invariant
Predicates which are true for every configuration of a run (cf. Definition 2.4.0.4). 67, 69, 70, 77, 78, 80, 125

Crash-Failure
failure model where processors simply stop their execution. 63, 190

deadlock
A configuration where no further step is possible (cf. Definition 2.3.1.4). 22, 24–26, 82, 136

Deterministic non-lifted Action
Deterministic Action defined over Localviews (cf. Definition 2.3.3.1). 37, 198

distributed algorithm
An algorithm designed to be processed by interconnected processes. Our definition of an distributed algorithm is based upon the definition of an algorithm and comprises further assertions (cf. Definition 2.3.2.3). iii, 2, 4, 5, 11–13, 15, 17, 35, 56, 63, 68, 71, 75, 77, 78, 83, 85, 86, 125, 128, 189, 191, 193

Distributed Consensus
A problem in distributed computing where the task is to make N processes agree on exactly one value. Therefore, an algorithm that solves Consensus is supposed to exhibit Validity, Agreement, Termination and Irrevocability. 5, 13, 68, 75, 78, 127, 136–138, 156, 172, 174, 189, 190, 192, 197, 199, 201, 213

distributed system
A distributed system $\mathfrak{D}_N$ is a collection of N individual computing devices (processes) that can communicate with each other (cf. Definition 1.2.0.1). iii, 1–3, 15–17, 20, 24, 40, 82, 85

DnA
Acronym for Deterministic non-lifted Action (cf. Definition 2.3.3.1). 37, 38, 43, 61–63, 73, 74, 95, 96, 100, 110, 132, 134, 143–146, 148, 162, 163, 190, 203

event
Trigger of an event-step. 11, 21, 22, 25, 34, 35, 39, 56, 73, 76, 77, 79–82, 86, 88, 90–93, 98, 128, 134, 151, 154, 169, 190–192, 198

event-step
Atomic step which happens unmotivated or at least without a dedicated process that triggered the event (e.g. the loss of a message while the message is transmitted)(cf. Definition 2.3.1.4). 21, 22, 56, 90, 92, 198, 200

Fail-Noisy
failure model where crashes can be detected but not always in an accurate manner. 64

Fail-Recovery
failure model where crashed processes can recover. 64, 190

Fail-Silent
failure model where processors also crash but their neighbours can detect this easily. 64

Fail-Stop
failure model where processors also crash but their neighbours can detect this easily. 64

failure detector
Abstraction that can be regarded as local modules monitoring the crash status of all involved processes in the system. 17, 20, 83, 127, 129, 130, 137, 138, 171

failure model
Model for the possible cases of failure. iii, 5, 16, 63–65, 157, 190, 192, 197, 198

fairness assumption
An assumption to guarantee progress for an algorithm.. 11, 25, 73, 83, 84, 155, 190, 191, 193, 197

final-stuttering
see FinSt. 23

function update
Concept to define a function with reference to another function (cf. Definition 2.3.2.1). 28, 34, 43

infinite run
An infinite sequence of configurations. (cf. Definition 2.3.1.5). 22, 24, 65, 155, 156

Irrevocability
Property of Distributed Consensus: Once a process decides on a value, it remains decided on that value.. 5, 68, 70, 198

Leader Election
A problem in distributed computing where the task is that exactly one process should output the decision it has become the leader [Lyn96]. 5

liveness
A liveness property is informally: a property asserting that something good eventually happens [Lam77] (cf. Definition 2.4.0.3). 4, 5, 12, 21, 64, 65, 67, 69, 70, 73, 75, 82, 83, 129, 172, 184, 190–193

Localview
The variables and resources a process can access. 30–35, 37, 38, 40, 62, 72, 131, 190, 198, 200, 203, 213

message history
Mapping which represents the messages in the system. 44, 82, 204

message passing
Communication method where processes send each other messages. iii, 40–42, 85, 127, 129, 174, 188, 190, 192

Mutual Exclusion
A problem in distributed computing where N cyclic processes enter and exit critical sections, the processes have to be programmed in such a way that at any one moment only one of theses processes is in its critical section [Dij65]. 5

non-lifted
A nonlifted action is an action that is defined on Localviews (instead of configurations). 30–34, 36–38, 73, 95, 132, 134, 145, 146, 151, 169, 184, 190, 204

process state
Registers, variables and program counter of a process. (cf. Definition 2.3.1.1). 19, 20, 31, 33–35, 38, 39, 42, 68, 71–73, 76, 77, 79, 128, 131, 171

process-action
Atomic transition executed by a process. 21, 22, 25, 28–30, 33, 35–38, 56, 62, 64, 73, 74, 76, 77, 88, 89, 91, 92, 95–98, 132, 134, 151, 169, 182, 184, 190, 213

process-step
Step taken by a process(cf. Definition 2.3.1.4). 21, 22, 30, 56, 90, 92, 113, 200

run
A sequence of configurations where each configuration and its successor are in the step relation (cf. Definition 2.3.1.6). 12, 16, 18–20, 22–26, 40, 51, 57, 59, 60, 65, 67–69, 76, 78–80, 82, 83, 125, 128, 129, 136, 138, 154, 174, 190, 191

Safe Algorithm
A formal specification of an algorithm which allows to verify safety properties (cf. Definition 2.3.1.3). 21–24, 79, 191, 197

safety
A safety property is informally a specification which states that something bad does not happen [Lam77] (cf. Definition 2.4.0.2). 4, 12, 13, 21, 64, 65, 67, 69, 70, 73, 75, 80, 129, 138, 156, 170–173, 184, 190–193

sequential algorithm
The behaviour of a single process defined step by step.. 1, 73

step
A transition which is either a process-step or an event-step (cf. Definition 2.3.1.4). iii, 16, 18, 20–22, 24–26, 32–34, 36, 43–46, 48, 50, 51, 53, 56, 73, 154, 170, 176, 177, 195–197

step relation
Relation containing all pairs of configuration $(\mathfrak{C}, \mathfrak{C}')$ where a step is possible from $\mathfrak{C}$ to $\mathfrak{C}'$ (cf. Definition 2.3.1.4). 21–23, 25, 200

Termination

Property of Distributed Consensus: All nonfaulty processes eventually decide.. 5, 68, 82, 185, 186, 198

Validity

Property of Distributed Consensus: If a process p_i decides a value v then there has to be a process p_j such that v has been input for p_j.. 5, 68, 198

Symbols

$\cdot$	Concatenation operator for sequences of configurations. 66, 67, 70
$\sqsubseteq$	Prefix relation for sequences of configurations. 66, 67, 70
$\mathfrak{A}$	An algorithm. 15, 21–25, 29, 31, 35, 39, 43, 47, 59, 68, 73, 75–80, 83, 89, 90, 97, 125, 135, 136, 152, 169, 184, 213
Σ	Set of all sequences. 22–24, 57, 65–68, 70, 78, 80, 83, 125, 205
bool	The set of boolean values. 8, 10, 21, 23–25, 30, 33, 34, 36, 37, 42, 47, 67, 68, 76, 77, 79–81, 83, 95, 125, 131
Correct	Set of all correct processes in the considered setting. 47, 65, 69, 136, 151, 153, 184, 185
C2LV	A function which returns for a configuration $\mathfrak{C}$ and process p_i the restricted view of p_i on $\mathfrak{C}$. 30–37, 40, 47, 56, 57, 59, 62, 71–73, 76, 95–97, 134–136, 148, 150, 152, 166, 169, 182, 184, 213
$\mathbb{C}$	Set of all configurations. 21–23, 25, 26, 30–37, 57, 62, 66, 67, 71, 72, 75–77, 79–81, 83, 185
$\mathbb{D}_{nA}$	Set of all DnAs. 37
Enabled	A predicate which is true if a certain action/event can be executed (cf. Definition 2.3.1.10). 25, 39, 184
FinSt	A run R with final-stuttering reaches a deadlocked configuration in time t and for all successive configurations $t' \geq t$ we repeat the configuration $R(t)$ (hence we have $R(t') = R(t)$). 23, 76, 199
$\mathbb{L}$	Set of all Localviews. 30, 31, 33, 37, 72, 95

Lift	Function that returns the lifted version of a non-lifted action. 33, 36, 37, 73, 95, 135, 151, 152, 169, 184
LV2C	A function which embeds a Localview of a given process into a configuration and returns the resulting configuration.. 31–37, 40, 43, 44, 47, 56, 57, 59, 62, 71–73, 76, 95–97, 135, 136, 148, 150, 152, 166, 169, 182, 184, 213
$\mathcal{M}$	Set of all messages. 27–29, 31, 32, 38, 42, 45, 47
msgs	All messages in the system (cf. Definition 2.3.5.1). 42, 44, 45, 47, 48, 101, 104, 105, 195, 196
lostmsgs	All messages in the system with status **lost**(cf. Definition 2.3.5.1). 47, 48, 105, 106
outMsgs	All messages in the system with status **outgoing**(cf. Definition 2.3.5.1). 42, 43, 45, 47, 48, 99, 102, 105, 195, 196
recmsgs	All messages in the system with status **received**(cf. Definition 2.3.5.1). 42–45, 47, 48, 99, 102, 105, 135, 148, 150, 195, 196
transitmsgs	All messages in the system with status **transit**(cf. Definition 2.3.5.1). 42–45, 47, 48, 99, 102, 105, 195, 196
$\mathbb{P}$	Set of all processes in the considered setting. 19–22, 25, 27–35, 37–39, 41, 49, 50, 52, 56, 57, 61, 62, 65, 68–70, 72, 75–80, 83, 92, 96, 109, 113, 127, 131–133, 136, 142, 145, 147, 151, 153, 155–157, 161, 165, 167, 170, 181, 183, 184, 186
$\mathrm{QPP}_{\mathcal{T}}$	Set of Tags used to describe the status of a message (without message loss). 41, 42
$\mathbf{Q}$	message history in a setting with quasi-reliable message passing. 41–48, 99–102, 104, 131, 132, 135, 141–143, 148, 150, 160, 195, 196
R	A run. 22, 23, 65, 203
$\mathcal{R}_{phs}$	Set of phases for Regular Registers. 52, 108
$\mathfrak{R}$	Set of status values for Regular Registers. 52, 61
$\mathcal{V}_{\mathrm{reg}}$	Set of all values that can be stored in the registers. 52–54, 56–63, 108, 113, 178
$\mathrm{view}^{\mathrm{r}}$	Readoriented view of a process on Regular Registers. 52–56, 59–61

$\mathrm{view}^{\mathrm{w}}$	Writeoriented view of a process on Regular Registers. 52–61, 108, 183
$\mathbb{S}$	Set of all process states. 20, 68, 71, 76, 77, 79
SumMsgs	Sum of all messages (cf. Definition 2.3.5.2). 44, 46, 102
SumOutgoing	Sum of all messages with status **outgoing** (cf. Definition 2.3.5.2). 44–46, 101, 102, 104
SumReceived	Sum of all messages with status **received** (cf. Definition 2.3.5.2). 44, 46, 101, 102, 104
SumTransit	Sum of all messages with status **transit** (cf. Definition 2.3.5.2). 44–46, 101, 102, 104
$\mathrm{Runs}_{\mathrm{ad}}$	Set of Admissible Runs. 24, 68
Σ^{ω}	Set of all infinite runs. 22
$\mathcal{T}_{\mathrm{ags}}$	Set of message status values. 27, 29, 32, 99
$\mathbb{T}$	A countably infinite set of points in time. 22, 23, 57, 65, 66, 76–81, 86, 136, 151, 153, 154, 170, 184–186
Ξ_{Ψ}	Subset of events which preserves a certain component. 76, 79
Ξ_{Φ}	Subset of actions which preserves a certain component. 76, 77, 79
Ξ	Subset of the actions and events which preserves a certain component. 76

Bibliography

[Agu10] Marcos K. Aguilera. Replication. chapter Stumbling over consensus research: misunderstandings and issues, pages 59–72. Springer-Verlag, Berlin, Heidelberg, 2010.

[Aro92] Anish Kumar Arora. *A foundation of fault-tolerant computing*. PhD thesis, Austin, TX, USA, 1992. UMI Order No. GAX93-09117.

[AS86] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1986.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing Fundamentals, Simulations, and Advanced Topics Second Edition*. 2004.

[CBM09] Bernadette Charron-Bost and Stephan Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

[CBS07] Bernadette Charron-Bost and André Schiper. The Heard-Of Model: Computing in Distributed Systems with Benign Failures. Technical report, 2007. Replaces TR-2006: The Heard-Of Model: Unifying all Benign Failures.

[CBTB00] Bernadette Charron-Bost, Sam Toueg, and Anindya Basu. Revisiting safety and liveness in the context of failures. In *Proceedings of the 11th International Conference on Concurrency Theory*, CONCUR '00, pages 552–565, London, UK, UK, 2000. Springer-Verlag.

[CDM11] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. Formal verification of consensus algorithms tolerating malicious faults. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *13th Intl. Symp. Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*, volume 6976 of *LNCS*, pages 120–134, Grenoble, France, 2011. Springer.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[Cho94] Ching-Tsun Chou. Mechanical verification of distributed algorithms in higher-order logic. In *Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 158–176, London, UK, 1994. Springer-Verlag.

[Cho95] Ching-Tsun Chou. Using operational intuition about events and causality in assertional proofs. Technical report, 1995.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43:685–722, July 1996.

[Cla78] Edmund M. Clarke. Proving the correctness of coroutines without history variables. In *ACM-SE 16: Proceedings of the 16th annual Southeast regional conference*, pages 160–167, New York, NY, USA, 1978. ACM.

[Cli73] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2:50–63, 1973. 10.1007/BF00571463.

[Cli81] M. Clint. On the use of history variables. *Acta Informatica*, 16:15–30, 1981. 10.1007/BF00289587.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, 1996.

[Cur58] Haskell B. Curry. *Combinatory Logic.* Amsterdam, North-Holland Pub. Co., 1958.

[DGM97] Marco Devillers, David Griffioen, and Olaf Muller. Possibly infinite sequences in theorem provers: A comparative study. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOL 97*, pages 89–104. Springer-Verlag, 1997.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8:569–, September 1965.

[FH07] A. Francalanza and M. Hennessy. A Fault Tolerance Bisimulation Proof for Consensus. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 395–410, 2007.

[FHB$^+$97] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Samuel Boutin, Eduardo Giménez, Gérard Huet, César Muñoz, Cristina Cornes, Judicaël Courant, Chetan Murthy, Catherine Parent, Christine Paulin-mohring, Amokrane Saibi, and Benjamin Werner. The coq proof assistant - reference manual version 6.1. Technical report, 1997.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[FMN07] Rachele Fuzzati, Massimo Merro, and Uwe Nestmann. Distributed Consensus, Revisited. *Acta Informatica*, 44(6):377–425, 2007.

[Fuz08] Rachele Fuzzati. *A formal approach to fault tolerant distributed consensus.* PhD thesis, Lausanne, 2008.

[Gaf98] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *In Proc of the 17th ACM Symp. Principles of Distributed Computing (PODC*, pages 143–152, 1998.

[GL00] Eli Gafni and Leslie Lamport. Disk Paxos. In *Distributed Computing*, pages 330–344, 2000.

[GR06] R. Guerraoui and L. Rodrigues. *Introduction to reliable distributed programming.* Springer-Verlag, 2006.

[GR07] Rachid Guerraoui and Michel Raynal. The alpha of indulgent consensus. *Comput. J.*, 50:53–67, January 2007.

[Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide, 1995.

[Gä99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31, 1999.

[Gä02] Felix C. Gärtner. Revisiting liveness properties in the context of secure systems. In *IN PROC. FASEC*, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hut94] Graham Hutton. Introduction to hol: a theorem proving environment for higher order logic by mike gordon and tom melham (eds.), cambridge university press, 1993, isbn 0-521-44189-7. *Journal of Functional Programming*, 4(04):557–559, 1994.

[JM05] Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs.* `http://afp.sf.net/entries/DiskPaxos.shtml`, June 2005. Formal proof development.

[Kin94] Ekkart Kindler. Safety and Liveness Properties: A Survey. In *EATCS Bulletin*, number 53, pages 268–272. June 1994.

[KNR12] Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms - from pseudo code to checked proofs. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2012.

[KWP99] Florian Kammüller, Markus Wenzel, and LawrenceC. Paulson. Locales a sectioning concept for isabelle. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–165. Springer Berlin Heidelberg, 1999.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lam85] Leslie Lamport. On interprocess communication, 1985.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[Lam01] Leslie Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, December 2001.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mcc] William Mccune. Otter 3.3 reference manual.

[McL96] John McLean. A General Theory of Composition for a Class of "Possibilistic" Properties. *IEEE Transactions in Software Engineering*, 22(1):53–67, 1996.

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[NF03] Uwe Nestmann and Rachele Fuzzati. Unreliable Failure Detectors via Operational Semantics. In *Advances in Computing Science - ASIAN 2003*, Lecture Notes in Computer Science, pages 54–71, 2003.

[Nip13] Tobias Nipkow. Programming and proving in isabelle/hol. Technical report, 2013.

[NPW] T. Nipkow, L. Paulson, and M. Wenzel. Isabelle's Logics: HOL.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Owi76] Susan Owicki. A consistent and complete deductive system for the verification of parallel programs. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 73–86, New York, NY, USA, 1976. ACM.

[Pau89] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, September 1989.

[Pau13] Lawrence C. Paulson. Isabelle's logics. Technical report, 2013.

[PC96] D. Peled and C. Chou. Formal verification of a partial-order reduction technique for model checking. In *In Proc. of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 241–257. Springer-Verlag, 1996.

[PLL97] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the paxos algorithm. In Marios Mavronicolas and Philippas Tsigas, editors, *WDAG*, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 1997.

[PNW03] Lawrence C. Paulson, Tobias Nipkow, and Markus Wenzel. Isabelle's logics: Fol and zf. Technical report, 2003.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22:299–319, 1990.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[VVK05] Hagen Völzer, Daniele Varacca, and Ekkart Kindler. Defining fairness. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 458–472. Springer, 2005.

[WDF$^+$09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 140–145, Berlin, Heidelberg, 2009. Springer-Verlag.

[Win03] Toh Ne Win. *Theorem-proving distributed algorithms with dynamic analysis.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, (Cambridge, MA), 2003.

List of Figures