# Modularity in the Design of Robust Distributed Algorithms

Giuliano Losa
giuliano.losa@epfl.ch

EPFL

# Abstract

A distributed system consists of several computers, connected by a network, that need to cooperate to accomplish a task. Distributed systems are now part of our everyday lives. From our means of transportation (cars contain dozen of computers) to our means of communication (the internet contains millions of computers), we must rely on distributed systems every day. Yet building trustworthy distributed systems is a challenge.

Distributed systems are especially hard to program because they need to constantly adapt their strategy to handle a changing environment: mobile components change location, communication links fail, servers crash and restart, users come and go, and attackers may try to break the system.

Experience has shown that it is not practical to combine several strategies ad-hoc, without strong guiding principles. The need to change strategy on the fly results in distributed algorithms that are so difficult to understand that they cannot be guaranteed to operate safely, let alone efficiently. Changing back and forth between only two different strategies is already a research challenge, but it is not sufficient given the wide variety of possible behavior of the environment. Adding more strategies quickly becomes intractable because every strategy must be able to pass the baton to every other, leading to a quadratic number of challenging cases.

This thesis proposes the Speculative Linearizability framework for building and reasoning about adaptive distributed algorithms in a practical way. The Speculative Linearizability framework soundly abstracts over the interaction between strategies, allowing each strategy to be designed, tested, and verified independently of the others. The abstraction guarantees that independently designed strategies are nevertheless compatible by construction: there is no need to check whether every strategy can pass the baton correctly to every other. By clearly separating each strategy, the Speculative Linearizability framework eliminates the complexity blowup that makes ad-hoc approaches impractical.

Our results have been been formalized in the TLA+ language and thoroughly tested with the TLC model checker. Moreover, we have proved and mechanically checked with the Isabelle/HOL interactive theorem prover that the core of our main result, the composition theorem, is correct.

**Keywords:** Distributed Computing, Adaptive Systems, Modularity, Speculation, Fault-Tolerance.

# Résumé

Un système informatique distribué est constitué des plusieurs calculateurs, communicant à travers un réseau, qui coopèrent pour mener à bien une tâche. Peut-être sans le savoir, nous dépendons tous les jours de systèmes informatiques distribués. Nos moyens de transports, voitures, avions ou trains, sont composés de dizaines voir de centaines de processeurs interconnectés. L'Internet est composé de milliers d'entités autonomes aussi bien physiquement que administrativement qui coopèrent pour nous offrir accès au web. Les "calculateurs" d'Internet, serveurs et autres routeurs, se comptent par millions.

Les systèmes distribués sont pourtant très difficiles à programmer car leur environnement est imprévisible : le système doit faire face à des pannes en tous genres, à des délais de communications, et au comportement imprévu des ses utilisateurs. Pour accomplir sa tâche efficacement dans un tel environnement, un algorithme distribué doit adapter sa stratégie aux changements de son environnement.

Les expériences passées ont montré qu'il n'est pas envisageable en pratique, sans fondations théoriques adéquates, de changer de stratégie dynamiquement lors du fonctionnement du système. Chaque stratégie doit être capable de passer le témoin à chaque autre stratégie. L'enchevêtrement qui résulte d'une telle combinaison de stratégies s'est avéré trop difficile à analyser pour en garantir le bon fonctionnement.

Cette thèse propose les fondations théoriques et une méthode, efficace en pratique, qui simplifient le développement des algorithmes distribués adaptatifs. La méthode présentée permet de développer, tester, et d'analyser chaque stratégie indépendamment des autres stratégies, tout en garantissant que les stratégies soient compatibles par construction. Il n'y a alors plus besoin de considérer l'enchaînement des stratégies. L'indépendance des stratégies les unes vis-à-vis des autres élimine la complexité du problème à sa source et permet, comme il est montré dans plusieurs exemples, d'obtenir des algorithmes adaptatifs très performants et sûrs.

Les résultats de cette thèse ont étés rigoureusement testés par Model Checking avec le logiciel TLC. De plus, le théorème à la base de nos résultats a été démontré mécaniquement avec le logiciel Isabelle/HOL.

**Mots clés :** Systèmes répartis, systèmes adaptatifs, modularité, systèmes spéculatifs, tolérance aux pannes.

# Acknowledgements

I would like to thank my advisors, Rachid Guerraoui and Viktor Kuncak, for giving me the opportunity to pursue a PhD and for their guidance towards its completion.

I thank the members of the jury, Joseph Sifakis, Tobias Nipkow, Stephan Merz, and André Schiper, for the time spent reviewing and evaluating my thesis and for helping me improve my thesis.

I thank all my colleagues and friends from EPFL for making my PhD more enjoyable. Let me cite a few, from the LPD: El Mahdi El Mhamdi, Swan Dubois, Nicolas Braud-Santoni, Mihai Letia, Victor Bushkov, Matej Pavlovic, Maxime Monod, Dan Alistarh, Florian Huc, Kristine Verhamme, Fabien Salvi; from the LARA: Eva Darulova, Hossein Hojjat, Etienne Kneuss, Philippe Suter, Ruzica Piskac, Mikael Mayer, Filip Konecny, Regis Blanc, Pierre-Emmanuel Cornilleau; from the LGG: Sofien Bouaziz, Mario Deuss, and Andrea Tagliasacchi.

Also many thanks to my friends outside EPFL: Martin and Mathias Lefort, Benjamin Provost, Yann Maillet-Contoz, the Wintsch Chalpin family, the Milani family, Nathalie Nowak, Kurt Jungbluth, Stefan Prennschütz-Schützenau, and my flatmates Julie Degand, Elsa Courdier, and Michael Ferez.

Special thanks to Gelare for her support during end of my PhD.

Finally, my biggest thanks go to my family, Bertrand, Pierre André, Roberto, Marie-Hélène, Gisèle, and Françoise and Jackie.

*Lausanne, Friday 7th February, 2014*

# Contents

# 1 Introduction

## 1.1 Robust Distributed Systems and Adaptation

Complex systems on which we depend on almost every day, like cars, airplanes, the electric grid, or the internet, contain dozens, hundreds, thousands, or even millions of computers. To deliver their services, these computers need to cooperate, forming what is called a *distributed system*: a system composed of multiple computers, spatially separated, that cooperate in order to achieve a collective goal.

The components of a distributed system behave according to a *distributed algorithm*, which assigns to each component an algorithm to execute. However, some aspects of a distributed system are not controllable and cannot be specified by an algorithm. For example, smart-phones change location, initiate communication, are turned on and off, etc. independently of the will of the network operator. Yet the cellular network must provide reliable service at all times. In the internet, routers and link may fail unexpectedly, users may start downloading files at any time, etc. Yet packets should be routed reliably at all times.

A distributed algorithm is usually said *correct* when it is *safe* and *live* [47], i.e., it never does anything wrong and it eventually delivers its service despite the unpredictable behavior of its components. For example, a cellular network may be said correct when users are eventually able to make a call when they request it (the system is live) and when a call never reaches the wrong number (the system is safe). The wide range of possible and uncontrollable behaviors makes the design of correct distributed algorithm especially challenging. However, correctness is not the only desirable property of a distributed algorithm. In practice, we often want a distributed system to have good performance, i.e., to deliver its service fast and not only eventually.

We say that a distributed algorithm is *robust* when the system consistently delivers good performance in all the varied conditions that it may encounter. Take the example of a road-traffic monitoring system that would use the GPS capability of smart-phones to build a real-time map of the traffic density. This system should provide timely information about the traffic on any road, regardless of it being rush hour, during which there is a high density of slow-moving users on the roads, or it begin a Sunday, when there are fewer users which move faster. Both situations are quite different. Let us think about how the algorithm running the system may gather traffic data. During rush-hour, the system must handle a lot of data. However, since cars move slowly and are densely concentrated, the algorithm could leverage the Wifi capability of smart-phones to gather the data using a gossip protocol, in which the information is propagated and aggregated from phone to phone before being sent, at a low frequency, to a server. Thanks to the gossip protocol, the algorithm would avoid overloading a central server. On Sunday there is less data to gather but the traffic is more fluid, causing unreliability in the Wifi communication between smart-phones: two cars will often get too far apart too quickly for the communication between phones to complete. Relying on the gossip protocol in this situation would bring the system to a halt. Instead, the algorithm could adapt to the situation and have the phones directly contact a central server through the cellular network.

The example of traffic monitoring shows that a robust distributed algorithm must *adapt* its strategy to the conditions that it faces. However, in many cases, there are dozens or more of possible conditions, instead of just two as in our example, and one can often not even forecast their existence, let alone provide for them, before the system is built. Therefore, one must be able to quickly add a new strategy to the algorithm, even though the system is already deployed and serving users. In other words, it must be possible to develop a robust distributed algorithm *incrementally*.

To sum up, we say that a distributed algorithm is *robust* when the following two conditions hold:

1. The distributed algorithm is able to *adapt* its strategy in response to change.
2. The distributed algorithm can easily be extended with new strategies, allowing *incremental* development.

However, achieving these two goals is challenging, intermingling performance and correctness issues.

## 1.2 The Problem of Dynamically Changing Strategy

There are two orthogonal aspects to adaptation: the *scheduling policy* and the *switching mechanism*. A scheduling policy determines when to change strategy and which new strategy to employ. A good switching policy would rely on accurate measurements of the execution and performance of the system and could apply control theory methods so as to maximize performance while maintaining stability, avoiding runaway oscillation of the system.

In contrast, a switching mechanism is an algorithm whose task is to bring about the changes dictated by the scheduling policy quickly and transparently to the users. The main issue faced by a switching mechanism is that changing the strategy of the whole system requires coordinated changes in all of its components while maintaining the functionality of the system to make adaptation transparent to the users. In this thesis we will study switching mechanisms, i.e. the problem of dynamically switching strategy without interrupting the functionality of the system. This problem is well-known [83, 78, 17, 93] but, with the exception of the Abstract framework [35], we are not aware of any systematic and general framework to address it.

In order to understand the switching mechanism problem, let us examine the case of State Machine Replication (abbreviated SMR) [54, 87]. SMR is a general technique used to build robust linearizable implementations of data types [57, 37]. SMR algorithms like Paxos [47] or PBFT [14], which are not adaptive, are notoriously hard to understand. The formal correctness proof of Disk Paxos [43] is about 7000 lines long. Only an informal proof, 35 pages long, of a simplified version of PBFT is known to the authors [13].

Adaptive SMR protocols are even harder. For instance, the Zyzzyva [46] protocol combines PBFT with a fast mode implemented by a simple agreement protocol. The fast mode is more efficient than PBFT when there are no failures. In the advent of failures, the fast mode cannot make progress and Zyzzyva falls back to executing PBFT. The ad-hoc composition of the fast mode with PBFT required deep changes to both algorithms and resulted in an entanglement that is hardly understandable. Moreover, Zyzzyva, being restricted to two modes, is very fragile [88]. If the common case is not what is expected by the fast mode one falls-back to PBFT, making the optimization useless. An adversary can easily weaken the system by always making it abort the fast mode and go through the slow one. Introducing a new strategy might make the protocol more robust but would require a new ad-hoc composition, including an alternative fast mode, at a cost comparable to the effort needed to build Zyzzyva from scratch, namely a Dantean effort. Given the diversity of situations encountered in practice, we are convinced that this ad-hoc approach is simply intractable.

Now consider the general case of implementing a specification with an adaptive algorithm that can dynamically switch between $n$ different modes. Despite changing mode, the algorithm must not violate the specification. Therefore, if each mode is built ad-hoc, there are $O\left(n^2\right)$ different switching cases in which correctness must be preserved across two different algorithms. Moreover, suppose that a new optimization is needed after the $n$ modes have been designed. Integrating a new mode means checking that changing from an existing mode to the new mode does not violate the specification, and vice versa. It may also be necessary to modify the existing modes to accommodate for the new one. In this situation, the interactions between any two modes may need to be reconsidered. When building algorithms with only two modes is a research challenge, such an approach is intractable.

The goal of the thesis is to simplify the development of robust distributed algorithms by proposing a theory of switching mechanisms, enabling a principled approach to the construction of adaptive algorithms.

## 1.3 Contributions

The thesis makes the following contributions:

1. We formalize the problem of devising robust adaptive algorithms. Our model abstracts over the scheduling policy, clarifying the task of the switching mechanism.
2. We propose a concrete solution, the Speculative Linearizability framework, to the problem of devising robust adaptive algorithms. The Speculative Linearizability framework soundly abstracts over the interaction between modes, allowing each mode to be designed, tested, and verified independently of the others. The abstraction guarantees that independently designed modes are nevertheless compatible by construction: there is no need to check whether every strategy can pass the baton correctly to every other.
3. We apply *Speculative Linearizability* to fault-tolerant message-passing algorithms, showing that state of the art algorithms, which are notoriously intricate, can be easily optimized with our framework, and to shared-memory algorithms, showing that Speculative Linearizability has a wide applicability.
4. We provide supporting material for others to use Speculative Linearizability to design new adaptive algorithms. The supporting material, consisting of TLA+ [53] specifications and Isabelle/HOL [76] theories, allows one to readily debug new adaptive algorithms with the TLC model checker and to obtain trustworthy correctness proofs of new adaptive-algorithms using Isabelle/HOL.

## 1.4 Speculative Linearizability

Speculative Linearizability is a correctness property simplifying the analysis of *speculative algorithms*. A speculative algorithm is an optimistic adaptive algorithm: a speculative mode behaves as if a particular assumption about the environment holds, achieving high performance if the assumption is true, but possibly failing otherwise. Different modes make different assumptions, thus, if a mode fails, another mode, whose assumption is speculated to hold, can take over the execution. When a mode fails, it is required to abort and switch to the next mode transparently to the users of the system. The problem of choosing the next mode so as to maximize performance is the task of the switching policy, which we do not consider in this work. In a nutshell, speculative algorithms are agile optimistic algorithms that favor failing fast and iterating rather than over-provisioning resources.

Examples of speculation include the Ethernet protocol, where processes speculatively occupy a single-user communication medium before backing off if collision is detected, or branch prediction in microprocessors, where the processor speculates that a particular branch in the code will be taken before discarding its computation if this is not the case. More recent instances of speculation include optimistic replication [46] or adaptive mutual exclusion [44]. In fact, most practical concurrent systems are speculative. In general, speculative systems may choose between a large number of modes, in order to closely match a changing environment.

In order to continue the execution after a mode failure, the two consecutive modes have to synchronize, using a switching mechanism that both mode understand. As we have seen in the example of SMR, designing algorithms which can abort and switch is very challenging. This is the problem that Speculative Linearizability addresses.

Speculative Linearizability builds on the notion of Linearizability [37, 51, 52], which already simplifies the development of distributed systems, but has no provision for adaptivity or speculation. The correctness of a system of processes communicating through linearizable objects reduces to the correctness of the sequential executions of that system. In other words, linearizability reduces the difficult problem of reasoning about concurrent data types to that of reasoning about sequential ones. In this sense, the use of linearizable objects greatly simplifies the construction of concurrent systems. At first glance, the design and implementation of linearizable objects themselves looks also simple. One can focus on each object independently, design the underlying linearizable algorithm, implement and test it, and then compose it with algorithms ensuring the linearizability of the other objects of the system. In short, linearizability is preserved by *inter-object composition*: a set of objects is linearizable if and only if each object is linearizable when considered independently of the others. However, the inter-object composition property does not help designing *robust* linearizable objects, which can switch between several modes.

Linearizable systems offer an interface composed of *invocation actions* and *response actions*. *Speculative linearizability* extends linearizability with the notion of *switch actions*, which makes it significantly richer than linearizability, yet it reduces to linearizability if these actions are ignored. Speculative linearizability augments classical linearizability with a new aspect of composition. Not only a system of concurrent objects can be considered correct if each of them is correct (*inter-object composition*), but a set of algorithms implementing different modes of the *same* object is correct if each mode is correct (*intra-object composition*). We express this new aspect through a new composition theorem. Intuitively, speculative linearizability captures the idea of *safe* and *live abortability*. A mode can abort if the assumptions behind speculation reveals wrong. When it does abort, it does so in a safe manner, by preserving the consistency (linearizability) of the object state. Moreover, the abort is also performed in a live manner, because a new mode can take over and make progress. Processes can switch asynchronously from one mode to another, without the need to wait for one another, as long as their execution, including switch actions, remains speculatively linearizable.

We apply Speculative Linearizability to the design of fault-tolerant data-type implementations in asynchronous message-passing systems. We present a speculatively-linearizable adaptive algorithm, $QZ$, which has the same progress guarantees as Generalized Paxos [49], a state of the art algorithm in the domain and a notoriously intricate algorithm, by combining two simple modes. Being speculatively linearizable, $QZ$ can be composed with any other speculatively-linearizable mode to boost its performance for new conditions, whereas Generalized Paxos is not easily extensible. Like Generalized Paxos, our algorithm can execute commuting requests in one message round-trip, a practical and common case. We also apply Speculative Linearizability to shared-memory consensus, obtaining a speculative shared-memory algorithm that uses only register if no processes contend on shared data structures. Our speculative shared-memory consensus algorithm demonstrates that Speculative Linearizability is also applicable in shared-memory.

## 1.5 Model Checking and Mechanically-Checked Proofs

The behavior of even modest distributed algorithm is often complex and contains many details that are notoriously easy to overlook, leading to bugs in implementations and errors in proof. To avoid making mistakes, we need the support of software tools that can test whether an algorithm has its intended properties or that can check our proofs. Therefore, we have formalized most of our work in TLA+, as well as the core property of Speculative Linearizability in Isabelle/HOL. The TLC model-checker allowed us to quickly explore new algorithms and debug them, while Isabelle/HOL allowed us to write mechanically-checked proofs. All of the algorithms presented in the thesis have been exhaustively model checked for small system sizes. A variant of Speculative Linearizability, exercising the principle at the core of Speculative Linearizability, has been proved correct in Isabelle/HOL [32].

The TLA+ and Isabelle/HOL specifications are one of contributions of the thesis, as they can be used by other researchers to quickly start debugging, with the TLC model checker, and proving, with Isabelle/HOL, new speculative algorithms.

## 1.6 Publications

Part of the work presented in this thesis has been published in the following three publications:

- Rachid GUERRAOUI, Viktor KUNCAK, and Giuliano LOSA. “Speculative Linearizability”. In: *PLDI*. Ed. by Jan VITEK, Haibo LIN, and Frank TIP. ACM, 2012, pp. 55–66. DOI: 10.1145/2254064.2254072.
- Rachid GUERRAOUI, Viktor KUNCAK, and Giuliano LOSA. “Abortable Linearizable Modules”. In: *The Archive of Formal Proofs*. Ed. by Gerwin KLEIN, Tobias NIPKOW, and Lawrence PAULSON. Formal proof development. http://afp.sf.net/entries/Abortable_Linearizable_Modules.shtml, 2012.
- Dan ALISTARH et al. “On the cost of composing shared-memory algorithms”. In: *SPAA*. Ed. by Guy E. BLELLOCH and Maurice HERLIHY. ACM, 2012, pp. 298–307. DOI: 10.1145/2312005.2312057

# 2 Specifying Distributed Systems

## 2.1 Introduction

Distributed algorithms are often very complex and some details of their structure and behavior are notoriously easy to overlook. To avoid mistakes, one can writing precise specifications of an algorithm and its properties in a formal specification language. Tools such as model checkers can then be used to test whether the algorithm satisfies its properties. In general, only a subset of all the behaviors of the algorithm can be explored by model checking. However, fully automatic model checkers can be easily used as debuggers of specifications. Writing a detailed formal proof can raise our confidence in the correctness of an algorithm beyond what is possible with a model-checker. However, only when a formal proof is *mechanically checked* by a computer can we reach the assurance that a distributed algorithm is correct.

This chapter is an introduction to the basic concepts of the theory of I/O automata and of the TLA+ language. In the rest of the thesis, we use the theory of I/O automata [61] for informal discussions and the TLA+ [53] language for formal specifications. In section 5.5, we describe the formalization and mechanical proof of one of our results in the Isabelle/HOL [76] interactive theorem prover.

Distributed algorithms can be concisely represented as the composition of several I/O automata because the components of a distributed system interact by performing *discrete joint actions* and otherwise evolve completely *asynchronously*. Composing two components represented as I/O automata results exactly in a system in which the two components, which are otherwise completely asynchronous, interact through specific discrete joint actions. Therefore, I/O automata composition accurately models the interaction between components of a distributed system.

In an effort to provide a trustworthy theory of adaptive distributed systems, we have formalized our work in the TLA+ language and we have checked the correctness of our results with the TLC model checker [94]. In section 2.4.6, we describe how to translate I/O automata specifications in TLA+ in order to use the TLC model checker.

There are many other specification frameworks targeting the description of distributed systems and their properties. Some frameworks are well-known as frameworks while others are better known by the name of their main component. Let us cite the BIP framework (Behavior, Interaction, and Priority) [6], Reactive Modules [4], Promela and the SPIN model checker [40], the NuSMV model checker [18], Bigraphical Reactive Systems [70], Abstract State Machines [10], and process calculi like CSP [39], the $\pi$-calculus [71, 72], and Petri nets [82].

In the rest of this chapter we present the theory of I/O automata, restricted to finite traces, and the TLA+ language. We also show how to express I/O automata specifications in TLA+, with the aim of checking them with the TLC model-checker.

Apart from section 2.4.6, which explains how to express I/O automata specifications in TLA+, the material presented in this chapter is well-known.

## 2.2 Notation

We now present the basic mathematical notions and notations that we will use throughout the thesis.

We will make use of basic mathematical expressions that should be familiar to the reader: quantified formulas, for example $\forall x \in S : P$ or $\exists x \in S : P$, set comprehensions, for example $\{x : P\}$ or $\{x \in S : P\}$, literal set expressions, as $\{e_1, \ldots, e_n\}$, and sequences, for example $\langle e_1, \ldots, e_n \rangle$. The cardinality of a set $S$ is denoted $Card(S)$.

If $es = \langle e_1, \ldots, e_n \rangle$ is a sequence and $i \in 1..n$, we write $es[i]$ for $e_i$ and $Last(e)$ for $e_n$. We use $\circ$ for sequence concatenation, $\langle e_1, \ldots, e_n \rangle \circ \langle f_1, \ldots, f_m \rangle = \langle e_1, \ldots, e_n, f_1, \ldots, f_m \rangle$. Appending an element $e$ to a sequence $es$ is denoted $Append(es, e)$. The set of all sequences of elements of a set $E$ is denoted $Seq(E)$. The length of a sequence $es$ is denoted $Len(es)$.

Arrays are multi-dimensional sequences. The elements at position $i, j$ of a two-dimensional array $A$ is denoted $A[i, j]$. Functions $F$ are the more general case of sequences and arrays whose domain is not restricted to integers.

We will often talk about the states $s$ of an automaton and about the components of $s$. We write $aComponent(s)$ for the component named $aComponent$ of the state $s$, and we omit the argument $s$ entirely when it is clear from the context.

## 2.3 I/O Automata

In this section we present the theory of I/O automata, restricted to finite executions. We use I/O automata as our main modeling framework throughout the entire thesis. Moreover, we have formalized a small part of the theory of I/O automata, restricted to finite executions, in Isabelle/HOL and we have used it to formalize some of our results. Our Isabelle/HOL theories can be found in appendix B.

I/O automata were first introduced by Lynch and Tuttle [61] to model asynchronous distributed systems. The theory of I/O automata is also described in details in chapter 8 of Lynch's book [59] , which contains many examples. In this section we give our own version of the theory of I/O automata, with some minor differences compared to Lynch and Tuttle. For example, the I/O automata of Lynch and Tuttle must be input-enabled whereas, to simplify specifications, ours do not.

An I/O automaton can be thought of as a *state-machine* plus an *interface*. First, an I/O automaton represent a system that has a state which is updated by taking discrete labeled actions. In this respect an I/O automaton is similar to what is often called a state machine or a traditional automata. Second, I/O automata have a *signature* which describes their interface and determines how two I/O automata synchronize when they are composed. Crucially, by using appropriate signatures, certain actions can be made internal to a component, in which case they will be executed completely asynchronously from the other components, and other actions, common to multiple components, can be matched and will be executed jointly, in a common discrete action, by all the components involved.

I/O automata conveniently describe distributed systems. A distributed system is usually composed of several components which interact through discrete transactions, or joint actions, and otherwise evolve independently. Given the characteristic of I/O automata composition, it is convenient to described distributed systems as the composition of several I/O automata representing the components of the system.

I/O automata can be used to describe a distributed system but also to specify at a high level of abstraction what a system should do. In other words, I/O automata can be used both for describing implementations and specifications.

In the rest of our work we will often need to prove that an implementation I/O automaton satisfies a specification I/O automaton. This means that the set of traces denoted by the implementation is a subset of the traces of the specification. We prove implementation using *forward simulations* or *refinement mappings* in conjunction with *history variables*.

Informally, proving by refinement that and I/O automaton $A$ implements and I/O automaton $B$ amounts to finding, for every step of $A$, a corresponding step of $B$ which has the same label. A refinement proof allows one to reason about the individual transitions of an I/O automaton and deduce a property of all its executions. Simulation proof techniques are reviewed in detail in Lynch and Vaandrager [62].

To simplify implementation proofs, one often introduces a sequence of intermediate I/O automata between the specification and the implementation and one shows using simulation proofs that, starting from the implementation, each I/O automaton implements the next in the sequence, up to the specification. For example, in section 3.4, we prove that the I/O automaton

$NDLin$ implements the I/O automaton $Lin$ in two steps, first showing that the I/O automaton $Lin'$ implements the $Lin$ I/O automaton, and then showing that $NDLin$ implements $Lin'$.

Finally, there are some tools that help devise and reason about distributed algorithms described using I/O automata. First, there is the Isabelle/HOLCF formalization of I/O automata theory [75, 74], which is available in the Archive of Formal Proofs. Second, there is the IOA Toolkit [42], which is composed of a formal specification of the IOA language, a simulator [92], a verifier based on the LP theorem prover [30], and a tool for generating Java programs from IOA specifications [31]. Unfortunately, many of those tools have not been maintained and there does not seem to be an active user community at the time of writing.

We will use the theory of I/O automata throughout the whole thesis, therefore we now formally define I/O automata and their related notions such as composition and simulations. Note that we deviate from the presentation of Lynch [59] on some details.

### 2.3.1 Definition of I/O Automata and their Traces

#### Signatures

A signature $sig$ is a triple consisting of three disjoint sets of *actions*, $Inputs\left(sig\right)$, the set of input actions of $Sig$, $Outputs\left(sig\right)$, the set of output actions, and $Internals\left(sig\right)$, the set of internal actions. The set of actions of a signature, denoted $Acts\left(sig\right)$, is the union of all three components, whereas the set of external actions, denoted $Ext\left(sig\right)$, is the union of the inputs and outputs. I/O-automata signatures are used to determine the result of composing several I/O automata, as we will see below.

#### State machines

A state machine $\Sigma$ is a tuple $\langle S, C, S_0, \delta\rangle$ where

- $S$ is the set of states of $\Sigma$;
- $C$ is the set of actions of $\Sigma$;
- $S_0 \subseteq S$ is the set of initial states of $\Sigma$;
- $\delta$ is the transition relation of $\Sigma$, which is a set of transitions $\left\langle s, a, s'\right\rangle$ where $s, s' \in S$ and $a \in C$.

The state machine $\Sigma$ is *deterministic* when it has a unique initial state and, for every state $s$ and action $a$, there is a unique transition $\left\langle s, a, s'\right\rangle \in \delta\left(\Sigma\right)$. When $\left\langle s, a, s'\right\rangle$ is a transition, we write $s \xrightarrow{a}_{\Sigma} s'$.

#### I/O Automata

An I/O automaton $A$ consists of a signature and a *state machine*. The set of actions of the state machine must be equal to the set of actions of the signature. We now consider an I/O automaton $A = \langle Sig, \Sigma \rangle$.

As shorthands, we write $Inputs(A)$ for $Inputs\left(Sig\right)$, $Outputs(A)$ for $Outputs\left(Sig\right)$, $Internals(A)$ for $Internals\left(Sig\right)$, $Ext(A)$ for $Ext\left(Sig\right)$, $Acts(A)$ for $Acts(A.sig)$, $Start\,(A)$ for $Start\,(\Sigma)$, $\delta\,(A)$ for $\delta\,(\Sigma)$, and $States\,(A)$ for $States\,(\Sigma)$.

Note that we do not require I/O automata to be input-enabled.

### Execution and schedules

We now define the notions of *execution fragment*, *execution*, and *schedule* of a state machine. The execution fragments, schedules, and traces of an I/O automaton are simply the ones of its state machine.

The *execution fragments* of a state machine $M$ are the sequences

$$\langle s_0, a_1, s_1, \ldots, a_n, s_n \rangle \tag{2.1}$$

where, for every $i \in 1..n$, $\langle s_{i-1}, a_i, s_i \rangle$ is a transition.

The *executions* are defined as the execution fragments whose first state is an initial state, $s_0 \in S_0$.

We say that an action $a$ is enabled in a state $s$ if there exists a transition, $\left\langle s, a, s' \right\rangle$, whose first state if $s$. We say that a state is *reachable* if there exists an execution of $\Sigma$ whose last state is $s$.

We define the *schedule* obtained from an execution $e$ as the projection of $e$ onto the actions, removing all states. The schedules of the state machine are the sequences $s$ such that there exists an execution $e$ whose schedule is $s$.

### Traces

The *trace* obtained from a schedule $s$ is the projection of $s$ onto the external actions. The traces of $A$ are the sequences $t$ such that there exists a schedule $s$ whose trace is $t$. We write $Traces(A)$ for the set of traces of $A$. When $e$ is an execution fragment, we define the trace of $e$, $Trace(e)$, as the trace of the schedule of $e$. Note that the trace of $e$ depends on the signature of the I/O automaton, whereas the schedule of $e$ does not.

We write $s \overset{t}{\Longrightarrow}_A s'$ when there exists an execution fragment $e$ whose first state is $s$, whose last state is $s'$, and such that $Trace\,(e) = t$.

### Implementation relation

We say that an I/O automaton $B$ *implements* an I/O automaton $A$, denoted $B \leq A$, when $A$ and $B$ have the same input actions, the same output actions, and the set of traces of $B$ is a subset of the set of traces of $A$.

## 2.3.2 Composition

### Signature composition

A set of signatures is said *compatible* when, for every two different signatures $Sig_1$ and $Sig_2$in the set, the outputs of $Sig_1$ and $Sig_2$ are disjoint and the internal actions of $Sig_1$ and $Sig_2$ are disjoint.

If $Sigs = \langle Sig_1, \ldots, Sig_n \rangle$ is a sequence of signatures where $\{Sig_1, \ldots, Sig_n\}$ is a compatible set of signature, then the composition of $Sigs$, $\prod_{i \in 1..n} Sig_i$, is such that

- The set of inputs of $\prod Sigs$ is the union of the set of inputs of the members of $Sigs$ minus the union of their sets of outputs,

$$Inputs\left(\prod Sigs\right) = \bigcup_{i \in 1..n} Inputs\left(Sigs[i]\right) \setminus \bigcup_{i \in 1..n} Outputs\left(Sigs[i]\right) \tag{2.2}$$

- The set of outputs of $\prod Sigs$ is the union of the set of outputs of the members of $Sig$.

$$Outputs\left(\prod Sigs\right) = \bigcup_{i \in 1..n} Outputs\left(Sigs[i]\right) \tag{2.3}$$

- The set of internal actions of $\prod Sigs$ is the union of the set of internal actions of the members of $Sig$.

$$Internals\left(\prod Sigs\right) = \bigcup_{i \in 1..n} Internals\left(Sigs[i]\right) \tag{2.4}$$

### I/O Automata composition

We say that a sequence of I/O automata is compatible when the corresponding sequence of signatures is compatible.

The composition of a sequence of I/O automata $\langle A_1, \ldots, A_n \rangle$, $\prod_{i \in 1..n} A_i$, is defined as follows.

- The signature of the composition is the product of the signatures $\langle Sig\left(A_1\right), \ldots, Sig\left(A_n\right) \rangle$.
- The states of the composition are the sequences $\langle s_1, \ldots, s_n \rangle$ where $s_i \in States\left(A_i\right)$ for every $i \in 1..n$.
- The initial states of the composition are the sequences $\langle s_1, \ldots, s_n \rangle$ where $s_i$ is an initial state of $A_i$ for every $i \in 1..n$.

– The transition relation of the composition is the set of transitions

$$\left\langle \langle s_1,\ldots,s_n\rangle, a, \langle s'_1,\ldots,s'_n\rangle \right\rangle \tag{2.5}$$

where if $a$ is an action of $A_i$, then $\langle s_i, a, s'_i\rangle$ is a transition of $A_i$, else $s'_i = s_i$.

We see that actions which belong to several components must be taken by all those components at once. Other actions are taken by their respective component while the other components remain unchanged.

Note that the traces of the composition of a compatible sequence only depends on the content of the sequence and not on the ordering of the I/O automata in the sequence. If $As$ and $Bs$ are two sequences of compatible I/O automata whose members are the same except for their ordering, then $\prod As$ and $\prod Bs$ have the same set of traces. Therefore, we will often talk about the composition of a set of I/O automata when we mean the composition of a sequences which contains exactly all the members of the set. Moreover, we write $A \times B$ for $\prod\langle A, B\rangle$.

We can also refactor nested composition of I/O automata.

**Lemma 2.1.** *Consider a two-dimensional array of I/O automata $Ass\,[i,j]$ where $i \in 1..n$ and $j \in 1..m$. Suppose that the members of $Ass$ are pairwise compatible, i.e., for every $i, j \in 1..n$ and $k, l \in 1..m$ where $i \neq j$ or $k \neq l$, $A_{i,k}$ and $A_{j,l}$ are compatible. Then, as far as traces are concerned, composing all the I/O automata of $Ass$ along the rows first is the same as composing along the columns first,*

$$Traces\left(\prod_{i\,\in\,1..n}\left(\prod_{j\,\in\,1..m} A_{i,j}\right)\right) = Traces\left(\prod_{j\,\in\,1..m}\left(\prod_{i\,\in\,1..n} A_{i,j}\right)\right) \tag{2.6}$$

### Monotonicity of composition

We can now state the first reduction theorem, which says that composition is monotonic with respect to the implementation relation: if $A_1 \leq B_1$ and $A_2 \leq B_2$ then $A_1 \times A_2 \leq B_1 \times B_2$.

**Theorem 2.1** (Monotonicity of Composition)**.** *If $\langle A_1,\ldots,A_n\rangle$ and $\langle B_1,\ldots,B_n\rangle$ are two compatible sequences of I/O automata and, for every $i \in 1..n$, $A_i \leq B_i$, then*

$$\prod\langle A_1,\ldots,A_n\rangle \leq \prod\langle B_1,\ldots,B_n\rangle. \tag{2.7}$$

This reduction theorem allows to reason about each component of a sequence independently and draw a conclusion about the composition of all the components.

### 2.3.3 Hiding and Projection

The $Hide(A, Acts)$ operators modifies the signature of the I/O automaton $A$ by removing all the actions of $Acts$ from the external signature of $A$ and transferring them to the internal actions of $A$. If $Sig$ is a signature, define

$$Hide(Sig, Acts) = \left\langle Inputs\left(Sig\right) \setminus Acts, Outputs\left(Sig\right) \setminus Acts, Internals\left(Sig\right) \cup Acts \right\rangle \quad (2.8)$$

Then we define $Hide(A, Acts)$ as the I/O automaton $A$ except that the signature of $Hide(A, Acts)$ is $Hide\left(Sig(A), Acts\right)$.

**Theorem 2.2.** *If* $A \leq B$*, then* $hide(A, S) \leq hide(B, S)$

The projection operator $proj(A, S)$ is defined in terms of hiding as

$$proj(A, S) = hide(A, Acts(A) \setminus S) \quad (2.9)$$

**Theorem 2.3.** *If* $A \leq B$*, then* $proj(A, S) \leq proj(B, S)$

### 2.3.4 Simulation Proofs

In this section we show how to prove that an I/O automaton $A$ implements an I/O automaton $B$ by using a refinement mapping in conjunction with history variables or by using a forward simulation. There are other types of simulation proofs, using prophecy variables or backward simulations. However we only use history variables and forward simulations in this thesis. For a thorough explanation of simulation proofs methods, we refer the reader to Lynch and Vaandrager [62].

We say that the I/O automaton $A_H$ is obtained by adding a history variable to the I/O automaton $A = \langle Sig, \langle S, S_0, C, \delta \rangle \rangle$ when there exist two nonempty sets $H$ and $H_0 \subseteq H$ such that

$$A_H = \langle Sig, \langle S \times H, S_0 \times H_0, C, \delta_H \rangle \rangle \quad (2.10)$$

where $\delta_H$ is such that

1. if $\left\langle \langle s, h \rangle, a, \left\langle s', h' \right\rangle \right\rangle$ is a transition of $\delta_H$, then $\left\langle s, a, s' \right\rangle$ is a transition of $\delta$;
2. if $\left\langle s, a, s' \right\rangle$ is a transition of $\delta$, then, for every $h \in H$, there exists $h' \in H$ such that $\left\langle \langle s, h \rangle, a, \left\langle s', h' \right\rangle \right\rangle$ is a transition of $\delta_H$.

**Theorem 2.4.** *If the I/O automaton* $A_H$ *is obtained from* $A$ *by adding a history variable then* $Traces(A_H) = Traces(A)$.

A refinement mapping from $A$ to $B$ is a *function* $f$ mapping states of $A$ to states of $B$ such that:

- if $s \in Start(A)$ then $f[s] \in Start(B)$;
- if $s$ is a reachable state of A and $s \xrightarrow{a}_A s'$, then
  - if $a \in Ext(B)$, then $f[s] \overset{\langle a \rangle}{\Longrightarrow}_B f[s']$;
  - if $a \notin Ext(B)$, then $f[s] \overset{\langle\rangle}{\Longrightarrow}_B f[s']$.

**Theorem 2.5.** *Consider two I/O automata $A$ and $B$ which have the same external signature. If $f$ is a refinement mapping from $A$ to $B$, then $A$ implements $B$.*

**Corollary 2.1.** *If the I/O automaton $A_H$ is obtained from $A$ by adding a history variable and there exists a refinement mapping $f$ from $A_H$ to $B$, then $A$ implements $B$.*

A forward simulation from $A$ to $B$ is a *relation* $r$ relating states of $A$ to states of $B$ such that:

- if $s \in Start(A)$ then $r[s] \cap Start(B) \neq \emptyset$;
- if $s$ is a reachable state of $A$, $s \xrightarrow{a}_A s'$, and $t \in r[s]$, then there exists a state $t' \in r[s']$ such that
  - if $a \in Ext(B)$, then $t \overset{\langle a \rangle}{\Longrightarrow}_B t'$;
  - if $a \notin Ext(B)$, then $t \overset{\langle\rangle}{\Longrightarrow}_B t'$.

**Theorem 2.6.** *Consider two I/O automata $A$ and $B$ which have the same external signature. If $r$ is a forward simulation from $A$ to $B$, then $A$ implements $B$.*

Forward simulations have the same power as the combination of a history variable and refinement mapping: one can prove that $A$ implements $B$ using a forward simulation if and only if one can prove it using a refinement mapping in conjunction with a history variable. A proof of this result appears in [62]. However, in practice, a proof may be easier with one or the other method. We will use theorem 2.6 and corollary 2.1 throughout the thesis to prove implementation relations between I/O automata. Backward simulations, not presented here, are formalized in the Isabelle/HOL theory called "Simulations" which can be found in appendix B.

## 2.4 TLA+

In this section we introduce TLA+ informally and we show how to translate I/O automata specifications in TLA+. Although we use the theory of I/O automata in the rest of the thesis, we have translated most of our specifications in TLA+ and we have used the TLC model checker to gain confidence in their correctness. Moreover, formal versions of the specifications found in the thesis are only given in TLA+, in appendix A.

There are already very good descriptions of TLA+, see for example the book Specifying Systems [53] or [69], and we would be unable to better explain TLA+. Therefore, instead of explaining TLA+ in details, we will only highlight its main features and give a few examples that we hope will suffice for the reader to understand our discussion. Note that the TLA+ examples are typeset with the TLA+ typesetter and do not follow the notation introduced earlier.

We have used TLC within the TLA Toolbox, which offers a user-friendly Integrated Development Environment for TLA+ specifications. The TLA Toolbox provides a graphical interface to edit, check, and prove specifications correct and the TLC model checker is integrated in the toolbox and allows fast and visual debugging of specifications. All the parameters of TLC can be control with the GUI and the graphical trace explorer simplifies the analysis of error traces. TLA+ specifications can be also be proved correct and mechanically checked in the TLA Toolbox with TLAPS [21]. However TLAPS is still in development at the time of writing and we have preferred using Isabelle/HOL for writing mechanically-checked proofs.

### 2.4.1 A Basic Example

TLA+ is a logic in which formulas denote sequences of states, called *behaviors*, in which each state is a function mapping *every* possible variable name (i.e. a string) to a value. A specification is just a formula.

Consider the following specification $Spec1$, where x is a variable:

$$Next1 \triangleq x' = x + 1$$

$$Init1 \triangleq x = 0$$

$$Spec1 \triangleq Init1 \land \Box Next1$$

Given a state $s$, we say that $s["x"]$ is the valuation of the variable $x$ in $s$. We say that $s$ is an *initial state* of $Spec1$ when $s$ satisfies $Init1$. We say that $\langle s, s' \rangle$ is a *step* or *transition* of $Spec1$ when the states $s$ and $s'$ satisfy $Next1$. Note that $Init1$ has no *primed* variable and that the second conjunct of $Spec1$ is of the form $\Box F$, where $\Box$ is the "always" operator of linear temporal logic and $F$ contains *primed* and unprimed versions of the variable $x$.

The formula $Spec1$ denotes the set of all the behaviors where

- the valuation of $x$ in the *initial state* is equal to 0, as described by $Init1$;
- for every step $\langle s, s' \rangle$, $s'["x"] = s["x"] + 1$ and all other variables *change arbitrarily*, as described by $Next1$. For example we could have $s["z"] = 42$ and $s'["z"] = "hello"$.

The formula $Spec1$ could specify a simple counter whose count is represented by the variable $x$.

### 2.4.2 The Implementation Relation

Consider the following specification $Spec2$.

$$Init2 \triangleq x = 0 \land y = \text{TRUE}$$

$$\begin{aligned} Next2 \triangleq\ & \land y' = \neg y \\ & \land \text{IF } y \text{ THEN } x' = x + 1 \text{ ELSE } \ x' = x \end{aligned}$$

$Spec2 \triangleq Init2 \land \Box Next2$

The formula $Spec2$ also specifies behaviors where $x$ is repeatedly increased by one. However, between two increments of $x$, there is one step in which only $y$ changes. Therefore, a behavior satisfying $Spec2$ does not satisfy $Spec1$. This is a problem because $Spec1$ and $Spec2$ could be descriptions of the same system, but at different levels of abstraction. In this case we would like to have a way of saying that $Spec2$ implement $Spec1$. As we have observed, one cannot define implementation as inclusion of the set of behaviours.

To define implementation in terms of trace inclusion we need to allow the specification $Spec1$ to "stutter", i.e., take steps where $x$ does not change while the other variables are updated arbitrarily. To obtain stuttering-invariant specifications in TLA+, one must write specifications in the form $Init \land \Box\,[Next]_{vars}$, where $Init$ constrains the initial state, $vars = \langle v_1, \dots, v_n \rangle$ is the list of all the variables appearing in the $Init$ or $Next$ formulas, and $[Next]_{vars}$ is defined as $Next \lor \left(v_1' = v_1 \land \dots \land v_n' = v_n\right)$.

Now reconsider our two examples, written in the form $Init \land \Box\,[Next]_{vars}$:

$Init1 \triangleq x = 0$

$Next1 \triangleq x' = x + 1$

$Spec1 \triangleq Init1 \land \Box[Next1]_{\langle x \rangle}$

$Init2 \triangleq x = 0 \land y = \text{TRUE}$

$$\begin{aligned} Next2 \triangleq\ & \land y' = \neg y \\ & \land \text{IF } y \text{ THEN } x' = x + 1 \text{ ELSE } x' = x \end{aligned}$$

$Spec2 \triangleq Init2 \land \Box[Next2]_{\langle x, y \rangle}$

In the new versions of $Spec1$ and $Spec2$, the behaviors satisfying $Spec2$ also satisfy $Spec1$. In TLA+, we can write this fact as the implication $Spec2 \Rightarrow Spec1$. Thus we can equivalently define the implementation relation as inclusion of behaviors, at the semantic level, or as implication, in the logic.

### 2.4.3 Refinement Mappings

We can prove that the specification $Spec2$ implements the specification $Spec1$ as follows. First, we prove that in all behaviors of $Spec2$, $x$ is a natural number and $y$ is a boolean. In TLA+, we state those properties as follows:

$Inv2 \triangleq x \in Nat \land y \in \text{BOOLEAN}$

$\text{THEOREM } Spec2 \implies \Box Inv2$

The formula $Inv2$ is called an invariant of the specification $Spec2$. The proof of the theorem is done by proving that the initial states of the specification satisfy the invariant and that if the invariant holds and one step is taken then the invariant holds again. In TLA+, we state it as follows, where priming a formula is like priming all its variables:

LEMMA $Init2 \implies Inv2$

LEMMA $Inv2 \land Next2 \implies Inv2'$

Second, we prove that the initial states of $Spec2$ are initial states of $Spec1$ and that if the invariant $Inv2$ holds of the first state of a step of $Spec2$, then this step is also a step of $Spec1$. This is called a *refinement proof.* In TLA+, it is formalized as follows.

THEOREM $Init2 \implies Init1$

THEOREM $Inv2 \land [Next2]_{\langle x, y \rangle} \implies [Next1]_{\langle x \rangle}$

The two theorems above imply that $Spec2 \Rightarrow Spec1$.

### 2.4.4 Hiding Internal State

Observe that if we look only at the $x$ variable, $Spec2$ and $Spec1$ have exactly the same behaviors. To make the observation formal we can hide the $y$ variable of $Spec2$, which we consider internal, using *temporal quantification.*

The specification $Spec2$ becomes

$Spec2 \triangleq \exists y : Init2 \land \Box[Next2]_{\langle x, y \rangle}$

Roughly speaking, the meaning of $Spec2$ is the set of all behaviors $b$ in which the valuation of $y$ of each state can be modified, obtaining $b'$, in order for $b'$ to satisfy $Init2 \land \Box[Next2]_{\langle x, y \rangle}$. This definition is not exactly the right one because one must take care that $Spec2$ remains stuttering-invariant.

We now have $Spec2 \Rightarrow Spec1$, as before, but also $Spec1 \Rightarrow Spec2$, formalizing the fact that $Spec1$ and $Spec2$ describe exactly the same behaviors when $y$ is hidden. Without hiding $y$, $Spec1 \Rightarrow Spec2$ does not hold because $y$ is unconstrained in $Spec1$.

### 2.4.5 Composing Specifications

Consider two specifications $F1$ and $F2$ of the form $F1 = Init1 \land \Box[Next1]_{vars1}$ and $F2 = Init2 \land \Box[Next2]_{vars2}$, where $vars1$ is the set of all the variables appearing in $F1$ and $vars2$ is the set of all the variables appearing in $F2$. The formula $F1 \land F2$ describes behaviors which satisfy both $F1$ and $F2$.

Suppose that $vars1$ and $vars2$ are disjoint. In this case the behaviors satisfying $F1 \wedge F2$ are composed of four kinds of steps: steps satisfying $Next1 \wedge Next2$, called *joint steps*, steps satisfying $Next1 \wedge vars2' = vars2$, steps satisfying $Next2 \wedge vars1' = vars1$, and steps satisfying $vars1' = vars1 \wedge vars2' = vars2$.

If $vars1$ and $vars2$ intersect, then every step modifying a variable of $vars1 \cap vars2$ must be a joint step. The specification of two communicating systems can therefore be obtained by conjoining two specifications that change common variables representing the interface between the two specifications. Note that, in the resulting specification, the two communicating components may take joint steps even when they do not communicate (when both only update variables not in $vars1 \cap vars2$). In contrast, two I/O automata in a composite I/O automaton take joint steps only when communicating.

This concludes our brief presentation of TLA+. We have not addressed many important topics, like using history and prophecy variables in refinement proofs, proving temporal properties, etc.. We refer the reader to [53, 69].

### 2.4.6 Expressing I/O Automata Specifications in TLA+

The TLC model checker allows to quickly debug specifications written in TLA+. Since we are primarily working with I/O automata, we needed to translate I/O automata specifications to TLA+ if we are to use the TLC model checker. Given an I/O automata specification, we would like to produce a TLA+ specification such that the sequences of states that satisfy the TLA+ specification are as similar as possible to the executions of the I/O automaton. In particular, we would like that an individual steps $\langle s, s' \rangle$ of a sequence of states satisfying the TLA+ specification map unambiguously to a state transition $\langle s_1, a, s_2 \rangle$ of the I/O automaton.

In this section we sketch a method for translating I/O automata specifications in TLA+. We have not followed this method strictly when producing the TLA+ counterparts to the I/O automata specification described in later sections, however the method exemplifies the basic ideas.

We have mainly used TLC to check that an I/O automaton $A$ implements a I/O automaton $B$. To do so, we must specify both $A$ and $B$ in TLA+, as formulas denoted $[\![A]\!]$ and $[\![B]\!]$, making sure that the transformation is sound, i.e., that $[\![A]\!] \Rightarrow [\![B]\!]$, in TLA+, implies that the I/O automaton $A$ implements the I/O automaton $B$. We assume that $A$ and $B$ have the same external signature; otherwise we already know that $A \leq B$ does not hold.

For simplicity, we assume that the components of the I/O automata that we consider, i.e., actions, states, initial states, and transition relation are expressed using the constant operators of TLA+, i.e., in a subset of TLA+ that excludes all temporal operators. Hence we assume that $[\![Sig(A)]\!] = Sig(A)$,$[\![Ext(Sig(A))]\!] = Ext(Sig(A))$, $[\![Internals(Sig(A))]\!] = Internals(Sig(A))$, $[\![States(A)]\!] = States(A)$, $[\![Start(A)]\!] = Start(A)$, and $[\![\delta(A)]\!] = \delta(A)$ are given.

The TLA+ specification $[\![A]\!]$ uses three variables $s_A$, $ext$, and $int_A$. The variable $s_A$ represents the state of $A$, the variable

$$ext \in [\mathit{flag} : \text{BOOLEAN}, act : [\![Ext\left(Sig\,(A)\right)]\!]] \tag{2.11}$$

is used to represent emitting an external action, and the variable

$$int_A \in [\mathit{flag} : \text{BOOLEAN}, act : [\![Internals\left(Sig\,(A)\right)]\!]] \tag{2.12}$$

is used to represent emitting an internal action. Similarly, the specification $[\![B]\!]$ uses the variables $s_B$, $ext$, and $int_B$, where $ext$ is shared with $[\![A]\!]$.

We use the operator

$$\begin{aligned} & Emit(A, a) \triangleq \\ & \quad \text{IF } a \in [\![Ext\left(Sig\,(A)\right)]\!] \\ & \quad \text{THEN } ext' = [\mathit{flag} \mapsto \neg ext.\mathit{flag}, act \mapsto a] \wedge int'_A = int_A \\ & \quad \text{ELSE } int'_A = [\mathit{flag} \mapsto \neg int_A.\mathit{flag}, act \mapsto a] \wedge ext' = ext \end{aligned} \tag{2.13}$$

to update the variables $ext$ and $int_A$, representing the I/O automaton $A$ emitting the action $a$. We use the flag to distinguish between stuttering and emitting the same action twice.

Finally, we define

$$\begin{aligned} [\![A]\!] \triangleq & \wedge s_a \in [\![Start\,(A)]\!] \\ & \wedge \Box\left[\exists a \in Acts\,(A) : Emit\,(A, a) \wedge \left\langle s_A, a, s'_A\right\rangle \in [\![\delta\,(A)]\!]\right]_{\langle s_A, ext, int_A\rangle} \end{aligned} \tag{2.14}$$

and, similarly, we define

$$\begin{aligned} [\![B]\!] \triangleq & \wedge s_a \in [\![Start\,(B)]\!] \\ & \wedge \Box\left[\exists a \in Acts\,(B) : Emit\,(B, a) \wedge \left\langle s_B, a, s'_B\right\rangle \in [\![\delta\,(B)]\!]\right]_{\langle s_B, ext, int_B\rangle} \end{aligned} \tag{2.15}$$

We conjecture that the statement $A \leq B$, in the theory of I/O automata, is equivalent to the following statement in TLA+:

$$(\boldsymbol{\exists}\, s_A, int_A : [\![A]\!]) \Rightarrow (\boldsymbol{\exists}\, s_B, int_B : [\![B]\!]) \tag{2.16}$$

Note how the state and internal actions of $A$ and $B$ are hidden, leaving only the variable $ext$, whose updates represent emitting external actions.

The transformation is simple but it is does not work well for I/O automata obtained as the composition of other I/O automata: we would like to define $[\![A \times B]\!]$ in terms of $[\![A]\!]$ and $[\![B]\!]$, for example as $[\![A]\!] \wedge [\![B]\!]$. However, taking $[\![A \times B]\!] = [\![A]\!] \wedge [\![B]\!]$ does not work: updates to the $ext$ variable have to be identical in both components or no update to $ext$ may happen, blocking all the actions $a \in (Ext(A) \cup Ext(B)) \setminus (Ext(B) \cap Ext(A))$.

To allow unilateral external actions, we separate the $ext$ variable in two variables $common$ and $ext_A$ in $[\![A]\!]$ and we separate the $ext$ variable in two variables $common$ and $ext_B$ in $[\![B]\!]$.

The three new variables allow $A$ or $B$ to take a step unilaterally, which represents emitting an internal action or an external action that is not common to both $A$ and $B$, or to take a joint step, which represent emitting an action common to $A$ and $B$.

When translating $A$, separating the variable $ext$ in the two variables $common$ and $ext_A$ requires knowing that $A$ will be composed with $B$. Therefore, we define the translation of the transition relation of $A$ in the context $B$, denoted $Next(A)_B$, as follows.

The formula $Next(A)_B$ uses the variables $ext_A$, $common$, $int_A$, and $s_A$. Define

$$\begin{array}{l}
Emit(A, a) \triangleq \\
\quad \text{IF } a \in [\![Ext(A)]\!] \\
\quad \text{THEN } \wedge\ int'_A = int_A \\
\quad \qquad\quad \wedge\ \text{IF } a \in Ext(A) \cap Ext(B) \\
\quad \qquad\qquad \text{THEN } common' = [flag \mapsto \neg common.flag, act \mapsto a] \wedge ext'_A = ext_A \\
\quad \qquad\qquad \text{ELSE } ext'_A = [flag \mapsto \neg ext_A.flag, act \mapsto a] \wedge common' = common \\
\quad \text{ELSE } int'_A = [flag \mapsto \neg int_A.flag, act \mapsto a] \wedge \text{UNCHANGED}\langle common, ext_A \rangle.
\end{array} \tag{2.17}$$

The operator $Emit(A, a)$ is used to update the variables $ext_A$, whose updates represent emitting an external action that is not common to $A$ and B, and the variable $common$, whose updates represent emitting an external action common to $A$ and $B$, and $int$, whose updates represent emitting internal actions of $A$.

Finally, define

$$\begin{aligned} & Next(A)_B \triangleq \exists a \in Acts(A) : \\ & \qquad \land Emit(A, a) \\ & \qquad \land a \notin Ext(B) \Rightarrow \text{UNCHANGED}\langle s_B, int_B, ext_B \rangle \\ & \qquad \land \left\langle s_A, a, s'_A \right\rangle \in [\![\delta(A)]\!] \end{aligned} \tag{2.18}$$

$$\begin{aligned} & Next(B)_A \triangleq \exists a \in Acts(B) : \\ & \qquad \land Emit(B, a) \\ & \qquad \land a \notin Ext(A) \Rightarrow \text{UNCHANGED}\langle s_A, int_A, ext_A \rangle \\ & \qquad \land \left\langle s_B, a, s'_B \right\rangle \in [\![\delta(B)]\!] \end{aligned} \tag{2.19}$$

$$vars \triangleq \langle s_A, int_A, ext_A, s_B, int_B, ext_B, common \rangle \tag{2.20}$$

$$\begin{aligned} & [\![A \times B]\!] \triangleq \\ & \qquad \land s_A \in [\![Start(A)]\!] \land s_B \in [\![Start(B)]\!] \\ & \qquad \land \Box\left[Next(A)_B \land Next(B)_A\right]_{vars} \end{aligned} \tag{2.21}$$

Note that we made sure that $A$ and $B$ cannot take a joint step except when they emit a common action.

If one want to check that $A \times B \leq C$, then the external variables of $C$ needs to be split so as to match $ext_A$, $ext_B$, and $common$.

Our method for translating composite I/O automata could be generalized to an arbitrary sequence of I/O automata but, as for the case of $A \times B$, the translation of each member of the sequence would depend on the signature of the other members of the sequence.

## 2.5 Conclusion

In this chapter we have presented the theory of I/O automata and the TLA+ language.

We have seen that I/O automata can describe distributed systems concisely thanks to a notion of composition which closely matches the behavior of distributed systems. Therefore we use I/O automaton in our informal discussion throughout the thesis.

In appendix A, we also precisely specify our results in the TLA+ language. The TLA+ specifications have been thoroughly model checked with the TLC model checker. The TLA+ specifications were obtained by translating our I/O automata specifications as described in section 2.4.6.

# 3 Linearizability: I/O-Automata Specification and Properties

## 3.1 Introduction

In this chapter we define the $Lin$ I/O automaton, which specifies *linearizability to a data type*. To ease later refinement proofs, we refine the $Lin$ I/O automaton to obtain the $NDLin$ I/O automaton. We also present the two well-known reduction theorems that simplify the development of linearizable distributed systems, and, finally, we relate our definition of linearizability to the original definition of Herlihy and Wing [37].

We define a model in which a set of *clients*, each a separate asynchronous process, access a data type D by calling a *local* interface: the interface of the data type is available locally at each client. Linearizability specifies the allowed behaviors of the implementation of the client's interfaces. Our I/O automaton specification can be seen as a reference implementation. However, how the interface is actually implemented is of no concern in this chapter.

Central to our I/O automaton definition of linearizability is the concept of *data-type representation.* A data-type representation is a state machine whose executions specify the sequential behavior of the data type. Crucially, the transition relation of a data-type representation can be minimized by grouping states that are in a certain equivalence relation. This property will be usefull in chapter 6 to optimize the execution of commuting requests in message-passing algorithms.

To ease future refinement proofs, we also present a more nondeterministic version of the I/O automaton specification of linearizability. The refinement will also showcase the use of the *idempotence* property of data-type representations.

The first reduction theorem is the *abstraction theorem* (theorem 3.4). It allows one to soundly abstract key parts of a distributed system from their inherent concurrent behavior, instead considering them sequential. This idea is formalized in the work of Filipolic et al. [28], which explains how and why a linearizable system is *observationally equivalent* to a simpler, sequential counterpart. We propose another version of the theorem, adapted to our setting, in section 3.5.

The second reduction theorem is the *inter-object composition theorem* (theorem 3.5). In contrast to the abstraction theorem, it concerns not the developers of a system who wish to use a linearizable component, but it concerns the designers of linearizable components. The inter-object composition theorem states that if a component $C_1$ is linearizable to a data type $D_1$ and a component $C_2$ is linearizable to a data type $D_2$, then the parallel composition of $C_1$ and $C_2$ is linearizable with respect to the parallel composition of $D_1$ and $D_2$. Therefore, one can reduce devising a linearizable implementation of a complex data type to devising several linearizable implementations of simpler data types.

We refer the readers to the works of Lamport [52], Herlihy and Wing [37], and Filipovic et al. [28] for more detailed discussions about linearizability and its properties. However, note that these works all rely on the traditional, trace-based, definition of linearizability, whereas our specification is an I/O automaton. The trace-based definition is presented in section 3.7.

## 3.2 Data Types and Data-Type Representations

### 3.2.1 Data Types

A data type describes the behavior of a system in which a set Π of clients invoke commands *sequentially*, i.e., a client invokes a command and receives a response before any other client can invoke a new command.

A data type $D$ consists of a triple $\langle C, O, \beta \rangle$, where C is the set of *commands* of the data type, where O is the set of *outputs*, and where $\beta$ is the set of behaviors of the data type.

Let $Req = \Pi \times C$ be the set of *requests*. A behavior is a sequence of *operations*, where an operation is a pair $\langle r, o \rangle$ consisting of a *request* r and of an *output* o. Note that our definition of a data type is slightly unusual because the requests contain a client identifier upon which the behavior of the data type may depend.

In the next subsection we define data-type representations. In the rest of the thesis we consider only data types which have a *deterministic*, *input-enabled*, and *idempotent* data-type representation. Unless otherwise noted, we consider such a data type $D = \langle C, O, \beta \rangle$.

### 3.2.2 Data-Type Representations

A data type may be represented by means of a state machine whose schedules specify the behaviors of the data type (see section 2.3.1 for the definition of state machines). Based on this observation, we now define the notion of *data-type representation*.

A data-type representation Δ of $D$ is a triple $\Delta = \langle \Sigma, O, \gamma \rangle$ consisting of a state machine $\Sigma = \langle S, C, S_0, \delta \rangle$, of the set of outputs $O$, and of an *output function* $\gamma$, which maps a state and a request to an output. The members of $S$ are called $\Delta$-states.

We say that a data-type representation is *deterministic* when the state machine $\Sigma$ is deterministic.

We say that a data-type representation is *input-enabled* when for every state $s \in S$ and for every request $r$, there exists a state $s'$ satisfying $\langle s, r, s' \rangle \in \delta$.

We now consider only deterministic and input-enable data-type representations. Therefore, we can define the following shorthands: we write $\perp$ for the unique state satisfying $S_0 = \{\perp\}$; we write $s \bullet r$ for the unique state $s'$ such that $\langle s, r, s' \rangle \in \delta$.

If $rs$ is a sequence of requests and $s$ is a state, we define $s \star rs$ as the final state obtained by executing all the requests of $rs$ in the order in which they appear, one by one:

$$s \star \langle\rangle = s; \quad s \star \langle r_1, \ldots, r_n \rangle = s \bullet r_1 \bullet \cdots \bullet r_n. \tag{3.1}$$

If $r$ is a request and $s$ is a state then $Contains(r, s)$ is true if and only if there exists a sequence of requests $rs$ containing $r$ such that executing $rs$ from the initial state results in $s$ ($\perp \star rs = s$).

### Idempotence

We say that the data-type representation $\Delta$ is *idempotent* when the two following properties hold.

**Property 1.** *A duplicate request leaves a $\Delta$-state unchanged: if $Contains(r, s)$ holds then $s \bullet r$ equals $s$.*

**Property 2.** *For every client $p$, if the last two requests of $p$ in a sequence $rs$ are the same, then they both produce the same output.*

Property 2 implies that the output of the last request of each client needs to be stored in the state to make later retrieval possible. As we will see in section 3.4 and chapter 5, property 1 will be useful in systems that might forget whether a request was executed or not. In this case one can just re-execute the request, obtaining the same output as before without impacting the execution of future requests. In practice, properties 1 and 2 can be implemented using timestamps to distinguish two otherwise equal requests, as in the example of a “set” data type in section 3.2.3. In the case of “one shot” data types like test-and-set and consensus, also presented in section 3.2.3, timestamps are not necessary.

Property 2 can be restated as follows: if one executes $\langle p, c \rangle$ before executing any number of requests not belonging to $p$, then re-executing $\langle p, c \rangle$ will result in the same output as the first time: if $rs$ is a sequence of requests such that for every request $\langle q, c' \rangle \in rs$, $q \neq p$, then

$$\gamma\left(s \star \left(rs \circ \langle p, c\rangle\right)\right) = \gamma\left(s, \langle p, c\rangle\right). \tag{3.2}$$

Using the first idempotence property, property 1, Property 2 can be simplified as follows. If $p$ and $q$ are two different clients, then the output obtained by executing $\langle p, c\rangle$ on $s$ is the same as the output obtained by executing $\langle p, c\rangle$ on $s \bullet \langle p, c\rangle \bullet \langle q, c' \rangle$,

$$\gamma\left(s \bullet \langle p, c\rangle \bullet \langle q, c' \rangle, \langle p, c\rangle\right) = \gamma\left(s, \langle p, c\rangle\right). \tag{3.3}$$

Let us take two short examples to illustrate idempotence. The transition relation represented in fig. 3.1 violates the first idempotence property because in state 2, after $r$ has been executed once, executing $r$ a second time should not change the state.

The transition relation represented in fig. 3.2 violates the second idempotence property supposing that $r_p$ is a request of the client $p$, $r_q$ is a request of the client $q \neq p$, and that $r_p$ does not produce the same output in state 1 and 3, $\gamma\left(1, r_p\right) \neq \gamma\left(3, r_p\right)$. There is no way to define $\gamma\left(4, r_p\right)$ without violating the second idempotence property. Once in state 4, one cannot know whether the last request of $p$ was execute in the upper path or in the lower path. Note that, for simplicity, both transition relations are not input enabled.

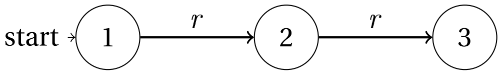

Figure 3.1 – A transition relation that violates the first idempotence property (property 1)

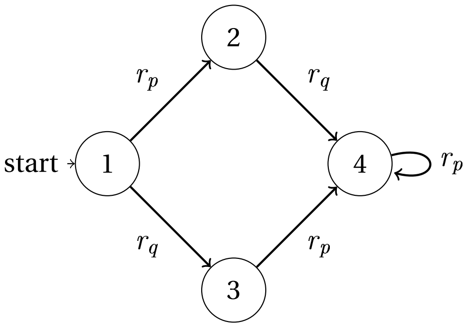

Figure 3.2 – A transition relation that violates the second idempotence property (property 2)

### Behaviors

The behaviors of $\Delta = \langle \Sigma, O, \gamma \rangle$, denoted $Beh\,(\Delta)$, are the sequences of the form $b = \langle op_1, \cdots, op_n\rangle$ such that there exists an execution $e = \langle s_0, r_1, s_1, \ldots, r_n, s_n\rangle$ of the state machine $\Sigma$ where

$$b = \left\langle \left\langle r_1, \gamma\left(s_0, r_1\right)\right\rangle, \left\langle r_2, \gamma\left(s_1, r_2\right)\right\rangle, \ldots, \left\langle r_n, \gamma\left(s_{n-1}, r_n\right)\right\rangle \right\rangle \tag{3.4}$$

The data-type representation $\Delta = \langle \Sigma, O, \gamma \rangle$ is a data-type representation *of* $D = \langle C, O, \beta \rangle$ when $Beh(\Delta) = \beta$. Note that a data-type representation uniquely determines a data type but that a data type may have several different representation.

In the rest of the thesis, and unless otherwise noted, we consider the data-type representation $\Delta$ of $D$, $\Delta = \langle \langle S, \langle \bot \rangle, C, \delta \rangle, O, \gamma \rangle$.

### 3.2.3 Examples of Data-Type Representations

In this section we present three examples of data-type representations which are deterministic, input-enabled, and idempotent.

**The Set Data Type**

The data type $Set(V)$ represents a set data structure containing members of the nonempty set $V$ and exposing the operations “add”, “remove”, and “contains”.

The commands of the $Set(V)$ data type are of the form $\langle$"add"$, v, ts\rangle$, $\langle$"remove"$, v, ts\rangle$, or $\langle$"contains"$, v, ts\rangle$, where $v \in V$ and $ts$ is a natural number that we call the time stamp of the command. The outputs of $Set(V)$ are booleans. The response to an “add” or “remove” operation is always true and the response to a “contains” operation indicates whether the queried element is in the set. Time stamps are used to detect duplicate requests: if the time stamp of a request from a client $p$ is smaller or equal to the last time stamp of $p$, the request has no effect and returns the value returned by the last operation of the invoking client.

A possible representation of $Set(V)$ is defined as follows. The set of state $S$ consists of three components:

1. the content of the set data structure;
2. for every client $p$,
   (a) the highest time stamp seen, $ts[p]$;
   (b) the output returned in response to the last request of $p$, $last[p]$.

The time-stamp and last-output components of the state are used to satisfy the two idempotence properties of data types.

In the initial state, the content is the empty set and, for every client, the time stamp is -1, which is lower than any time stamp that may appear in a request, and the last output is arbitrary.

The transition relation $\delta$ changes the state as follows. For every request of a client $p$, the time stamp $ts$ of the request is checked and, if it is lower than or equal to $ts[p]$, then the state is left unchanged. If $ts$ is strictly greater than $ts[p]$, then $ts[p]$ is set to $ts$. Moreover, a command $\langle$"add"$, v, ts\rangle$ adds $v$ to the members of the set, a command $\langle$"remove"$, v, ts\rangle$ removes $v$ from the set, and a command $\langle$"contains"$, v, ts\rangle$ leaves the state unchanged.

Given a request of the client $p$ with time stamp $ts \leq ts\left[p\right]$, the output function $\gamma$ always returns the value of $last\left[p\right]$ . Otherwise, if the addition or removal of an element is requested, then true is returned, and if the request is of the form $\langle$"contains"$, v, ts\rangle$, $\gamma$ returns true if $v$ is a member of the set and false otherwise.

### The Consensus Data Type

We now specify a consensus data type that will allow us to later define the consensus problem as the problem of linearizability to the consensus data type.

The commands of $Cons\left(V\right)$ are of the form $\left\langle \text{"propose"}, v\right\rangle$ and the outputs are of the form $\langle$"decide"$, v\rangle$, where $v \in V$. In every behavior of the consensus data type, the argument $v_1$ to the first request is the value which is decided upon: all requests return $\langle$"decide"$, v_1\rangle$.

The consensus data type $Cons\left(V\right)$ may be represented as follows. We assume that there are at least two different values in $V$, otherwise consensus is trivial. Let the set of states be the set $\left\{V\right\} \cup V$, where $V$ means that no value has been chosen yet and $v \in V$ means that the value $v$ has been chosen. In the initial state, no value has been chosen ($\bot = V$).

The transition relation $\delta$ is such that the first value proposed is chosen,

$$V \bullet \left\langle \text{"propose"}, v\right\rangle = v, \tag{3.5}$$

and if a value is already chosen, then the same value is still chosen,

$$\left\{v\right\} \bullet \left\langle \text{"propose"}, v'\right\rangle = v. \tag{3.6}$$

The transition function $\delta$, when $V = \left\{v_1, v_2\right\}$, is represented graphically in fig. 3.3.

The output function $\gamma$ returns the chosen value, i.e., if the state is $V$, then $\gamma$ returns the argument of the propose request, and if the state if of the form $v \in V$, then it returns $v$, the chosen value.

Note that the representation is idempotent, but it does not use time stamps. We will later see that linearizability to this data type is equivalent to the traditional formulation of the consensus problem.

### The Test-and-Set Data Type

In the same vein as for consensus, the $TestAndSet$ data type can be represented without the use of time stamps.

The $TestAndSet$ data type has only one command “ts” and returns either “Won” or “Lost”. Its behaviors are such that the first client to invoke the command “ts” wins and all the others loose.

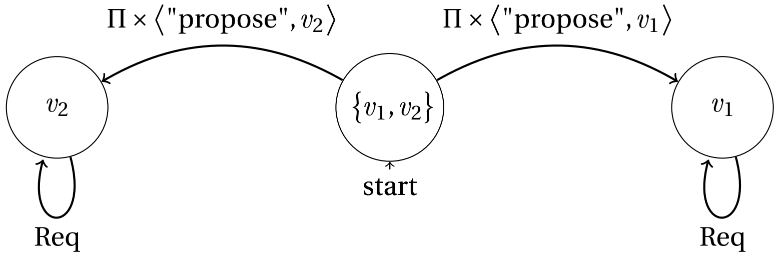
$\Pi \times \langle$"propose"$, v_2\rangle$
$\Pi \times \langle$"propose"$, v_1\rangle$
$v_2$
$\{v_1, v_2\}$
$v_1$
start
Req
Req

Figure 3.3 – The transition relation of a representation of the consensus data type when $V = \{v_1, v_2\}$

To ensure that the winner gets the response “Won” even if it invokes the “ts” command twice or more, the state needs to contain the identity of the winner. Therefore we let the state be either the full set of clients Π, indicating that no client won, or a single client $p$, indicating that $p$ won. The initial state is of course Π.

The transition relation leaves the state unchanged if the state is of the form $p \in \Pi$ and otherwise, if the state is Π, sets the state to the identity of the client which invoked the command. The transition relation, when $\Pi = \{p_1, p_2\}$, is represented in fig. 3.4.

The output function $\gamma$ always returns “Won” in the state Π. In a state $\{p\}$, $\gamma$ returns “Won” if the issuer of the request is the client $p$ and “Lost” otherwise.

Note that the *TestAndSet* data type is idempotent.

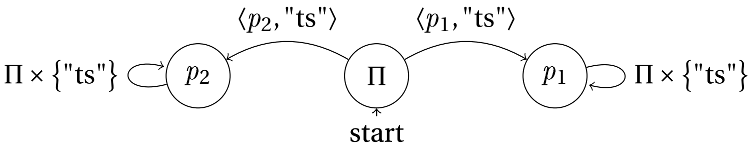

Figure 3.4 – The transition relation of a representation of the *TestAndSet* data type, when $\Pi = \{p_1, p_2\}$

### The Generic Data Type

The *Generic* $(C)$ data type takes its set of commands $C$ as parameter and, given a request $r$, it returns the complete sequence of requests that it has received so far except that duplicates are removed, called its execution history. In case of a duplicate request, the output is the execution history truncated at the previous occurrence of the duplicate.

A possible representation of the *Generic* $(C)$ data type would maintain the current history in its state, starting from the empty sequence, and would execute a command $c$ by appending $c$ to the current history, which it then returns. Thus the *Generic* data type returns, in response to every request, its complete execution history. For idempotence, a request is appended only if it does not yet appear in the sequence of requests executed so far. Moreover, the output to a duplicate request is the prefix of the execution history which ends with the duplicate request.

Formally, consider the execution history $h \in Seq(Reqs)$. If $r$ appears in $h$, then $\delta(h,r) = h$ and $\gamma(h,r)$ is a prefix of $h$ ending with $r$, else, if $r \notin h$, $\delta(h,r) = \gamma(h,r) = Append(h,r)$.

We will mainly use the $Generic$ data type to model check our specification with the TLC model checker.

### 3.2.4 Space of Possible Representations

A given data type has several possible representations, which differ in their state space and in the shape of their state-transition graph. Changing representation can be useful to prove the linearizability of an algorithm by refinement. Indeed, our I/O automata specification of linearizability (section 3.3) is parameterized by a data-type representation. Choosing a data-type representation whose structure is similar to the algorithm being proved can ease the proof. Notably, in chapter 6, we will use the history data-type representation (section 5.3.1), which "folds" commutative operations, in order to analyse algorithms that optimize the execution of commutative operations.

We have assumed that the data type D has a deterministic, input-enabled, and idempotent representation $\Delta = \langle\langle S, C, \{\bot\}, \delta\rangle, O, \gamma\rangle$. To give an idea of the range of possible data-type representations of $D$ we define two representations based on $\Delta$. The first, $Unfold(\Delta)$, has a state space of maximal cardinality. The second, $Fold(\Delta)$, has a state space of minimal cardinality.

The representation $Unfold(\Delta)$ is similar to the $Generic$ data type, defined in the preceding section, in that its state contains the execution history, i.e., the full sequence of requests that have been received so far. However, in contrast to the$Generic$ data type, responses are not histories, but are outputs computed by executing the entire history.

Formally, define $Unfold(\Delta) = \langle\langle S_1, C, \{\bot_1\}, \delta_1\rangle, O, \gamma_1\rangle$ where $S_1$ is the set of all histories, $Seq(Req)$, where the initial state $\bot_1$ is the empty history, $\langle\rangle$, where $\delta_1(s,r)$ appends $r$ to the history $s$, and where the output $\gamma(s,r)$ is obtained by executing, using the transition function of $\Delta$, the history $s$ starting from the initial $\Delta$-state, obtaining $\gamma(\bot \star s, r)$.

In contrast to $Unfold(\Delta)$, in which there is a one to one mapping from sequence of requests to states, the representation $Fold(\Delta)$ merges all the states that can possibly be merged. We say that two states of $\Delta$ are *output equivalent* if they cannot be distinguished by executing requests and looking at the output produced,

$$s \equiv s' \Leftrightarrow \forall rs \in Req^*, r \in Req : \gamma(s \star rs, r) = \gamma(s' \star rs, r). \tag{3.7}$$

Note that the output equivalence relation on states is reflexive, symmetric, and transitive, therefore we can define its equivalence classes, which form a partition of the set of states.

Let us write $Eq(s)$ for the equivalence class of a state $s$. We now define $\delta'$ and $\gamma'$ such that $\delta'\left(Eq(s), r\right) = Eq(\delta(s,r))$ and $\gamma'\left(Eq(s), r\right) = \gamma(s,r)$. The functions $\delta'$ and $\gamma'$ are well defined because all the members of an equivalence class are output equivalent, by definition.

We now define $Fold(\Delta) = \langle\langle\{Eq(s) : s \in S\}, C, \{Eq(\perp)\}, \delta'\rangle, O, \gamma'\rangle$.

Note that $Fold(\Delta)$ minimizes the number of state that a representation of $D$ may have.

## 3.3 I/O-Automaton Specification of Linearizability

In this section we define the I/O automaton $Lin(\Delta)$, which is our specification of linearizability. We say that an I/O automaton A is linearizable to $D$, or is a linearizable implementation of $D$, when $A$ implements $Lin(\Delta)$. This definition of linearizability is equivalent the original definition, which is presented in section 3.7. A TLA+ version of the specification presented in this section appears in appendix A.

We begin, in section 3.3.1, by defining the concept of *well-formed data-type implementation* using an I/O automaton. This definition provides a simple example of the kind of I/O-automata specification that we use throughout the thesis.

### 3.3.1 Well-Formed Data-Type Implementations

In the preceding section we have defined data types. A data type specifies a set of sequences of operations, where each operations is constituted of a request and a response.

However, a data type is not a description of a distributed system. In a distributed system, operation may not be considered atomic: responding to a request often requires coordination among the clients. Thus a model of a distributed system should consider the invocation of a request and the production of an output as two separate events. Moreover a distributed system implementing a data type will be used by other components of a bigger application. Thus we need a notion of interface and composition.

In this section we define the $Seq(D)$ I/O automaton, which specifies the interface that a data-type implementation should offer and whose traces are those produced by a set of *asynchronous sequential processes.* We say that the traces of $Seq(D)$ are the *well-formed* traces.

An implementation of the data type $D$ offers the interface of $D$ *locally* to each member of a set $\Pi$ of sequential clients, treating invocations and responses as separate actions. Each client may locally *invoke* the data type with a command and later receive a *response* containing an output. We stress that invocations and response are *local,* meaning that no communication across different agents is necessary to make or receive calls through the interface.

The *invocation actions* $a$ consist of an invoking client, denoted $Proc(a)$, and a command, denoted $Cmd(a)$. The invocation of command $c$ by client $p$ is denoted $Inv_p(c)$. The set of all invocation actions is denoted $Invs$ and the set of all invocation actions of a client $p$ is denoted $Invs_p$.

The *response actions* $a$ indicate the client which receives the response, noted $Proc(a)$, and an output, denoted $Output(a)$. The response to client $p$ with output $o$ is denoted $Resp_p(o)$. The set of all response actions is denoted $Resps$ and the set of all response actions of a client $p$ is denoted $Resps_p$. Note that the sets $Invs$, $Resps$, $Invs_p$, and $Resps_p$ depend on the data type $D$.

It will latter be useful to project a trace $t$ of invocations and responses onto the actions of a particular client, denoted $t|p$.

As we have said earlier, we assume that the clients Π are *sequential* and execute *asynchronously* from each other. A client is sequential when, after invoking a request, the client waits for a response before invoking a new request, and when only one response may appear in between two invocations. The clients are purely asynchronous when there is no dependency between their respective behavior. The I/O automaton *Seq* formalize these requirements.

We define *Seq* as the composition of the I/O automata $Seq(p)$, for every client $p \in \Pi$,

$$Seq = \prod_{p \in \Pi} Seq(p). \tag{3.8}$$

Every trace of the I/O automaton $Seq(p)$ starts with an invocation and continues with alternating responses and invocations, modeling a sequential client. The state machine of $Seq(p)$, which realizes this behavior, simply tracks the control flow location of the client $p$, namely "ready" or "pending". In the initial state, every client is "ready". Then, $Seq(p)$ executes as follows.

1. An invocation action $Inv_p(c)$ is enabled when the client $p$ is "ready" and changes the control flow location to "pending".
2. A response action $Resp_p(o)$ is enabled when the client $p$ is "pending" and changes the control flow location to "ready".

The transition relation of the I/O automaton $Seq(p)$ is represented graphically in fig. 3.5.

To understand what the composition $\Pi_{p \in \Pi} Seq(p)$ does, we also need to know the signatures of the $Seq(p)$ I/O automata. The inputs of $Seq(p)$ are the invocation actions of $p$, $Invs_p$, the outputs of signature of $Seq(p)$ are the response actions of $p$, $Resps_p$, and $Seq(p)$ has no internal actions. Note that if $p \neq q$, then $Seq(p)$ and $Seq(q)$ have no actions in common. Their composition is therefore purely asynchronous.

By definition of I/O automata composition and of the signature of $Seq(p)$, the inputs of the I/O automaton *Seq* is the union of the inputs of the $Seq(p)$ I/O automata, namely the set of all invocation actions $Invs$, and the outputs of the I/O automaton *Seq* is the union of the outputs of the $Seq(p)$ I/O automata, namely the set of all response actions $Resps$.

Finally, we say that an I/O automaton $A$ is a *well-formed* distributed implementation of the data type $D$ when $A$ implements the I/O automaton $Seq$. We also say that a trace $t$ is *well-formed* when $t$ is a trace of $Seq$.

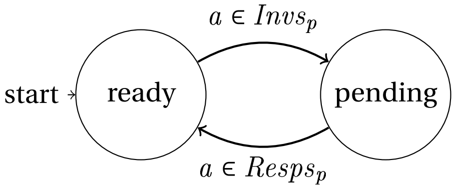

Figure 3.5 – The transition relation of the I/O automaton $Seq(p)$.

### 3.3.2 The Linearizability I/O Automaton

In this section we define the I/O automaton $Lin\,(\Delta)$, or $Lin$ for short, and we say that an I/O automaton $A$ is *linearizable* to D when there exists a data-type representation $\Delta$ of $D$ such that the projection of $A$ onto the invocation and response actions, denoted $\pi_{i/r}\,(A)$, implements $Lin\,(\Delta)$.

In fact the set of traces of the I/O automaton $Lin\,(\Delta)$ is the same for every representation $\Delta$ of D (theorem 3.1). However, choosing an appropriate data-type representation can make refinement proofs easier.

Let us now describe the $Lin$ I/O automaton. Consider a well-formed trace $t$. Let us say that a request $r$ is pending at some position $i$ in $t$ when the request has been invoked at a position $j < i$ but has not received a response before position i. For example, in an execution of the $Seq$ I/O automaton, when a component $Seq\,(p)$ is in the state “pending”, then there is a request $\langle p, c\rangle$ of client $p$ which is pending. We say that a request $r$ is pending in $t$, with no mention of a position, when $r$ is pending at the last position of $t$.

The I/O automaton $Lin$ is a well-formed data-type implementation of $D$: The external interface of the $Lin$ I/O automaton is the same as the one of the $Seq$ I/O automaton and the set of traces of the $Lin$ I/O automaton is a subset of the set of traces of the $Seq$ I/O automaton.

The $Lin$ I/O automaton uses the data-type representation $\Delta$, internally, to determine the output to the requests that it receives. The states of the $Lin$ I/O automaton consist of four components:

1. $dState$, tracking the current $\Delta$-state;
2. for every client $p$,
   (a) $status\,[p]$, tracking the control flow location of $p$;
   (b) $pending\,[p]$, containing the pending request of $p$;
   (c) $nxtOut\,[p]$, containing the next output that should be sent to $p$ in a response.

The control flow location $status[p]$ of the client $p$ can be either "ready", "pending", or "linearized". Initially, every client is ready and the value of $dState$ is $\perp$.

An $Inv_p(c)$ action is enabled when $status[p]$ is "ready". Its effect is to update $status[p]$ to "pending" and to update $pending[p]$ to $\langle p, c\rangle$. In order to produce a response, the client must first reach the status "linearized", by executing a $Linearize_p$ action.

The $Linearize_p$ action is enabled when $p$ is in status "pending". Its effect is to update the status of $p$ to "linearized", to update $dState$ by executing the pending request of $p$, setting $dState$ to $dState \bullet pending[p]$, and to update $nxtOut[p]$ to the output obtained by executing the pending request of $p$ on $dState$, $\gamma(dState, pending[p])$. We say that $pending[p]$ has been linearized. The $Linearize_p$ actions, for $p \in \Pi$, are the only internal actions of the I/O automaton $Lin$.

A $Resp_p(o)$ action is enabled if the client $p$ is in status "linearized" and if the output $o$ is equal to the output that was computed by the preceding $Linearize_p$ action, which is found in $nxtOut[p]$.

The control flow of a client $p$ in the $Lin$ I/O automaton is represented graphically in fig. 3.6.

We see that a $Linearize_p$ action must happen at some point in between every invocation-response pair, and that, to the client observing its external interface, it will appear as if its request was executed on $\Delta$ at some point in between the invocation and the response. Therefore, if the operations of two clients $p_1$ and $p_2$ overlap, then their requests, denoted $r_1$ and $r_2$, may be executed in the order $r_1, r_2$ or in the order $r_2, r_1$. However, if the operations do not overlap, for example when $r_2$ is invoked after $p_1$ received a response, then only one execution order is possible, $r_1, r_2$ in this case.

The TLA+ version of the specification of the I/O automaton $Lin$ can be found in appendix A.

**Theorem 3.1.** *If $\Delta_1$ and $\Delta_2$ are two representations of D, then $Lin(\Delta_1)$ and $Lin(\Delta_2)$ have exactly the same set of traces.*

*Proof.* Because all representation of D have the same set of behaviors. □

### 3.3.3 Examples: consensus and test-and-set

Consider the $TestAndSet$ and the $Consensus$ data types that we have defined in section 3.2.3. Implementing the I/O automaton $Lin(TestAndSet)$ is equivalent to solving the test-and-set problem in its usual formulation, and implementing the $Lin(Consensus)$ is equivalent to solving consensus in its usual formulation.

Let us look into more details to the case of consensus. The consensus problem is usually formulated as follows. Each client proposes a value and must subsequently decide on a value,

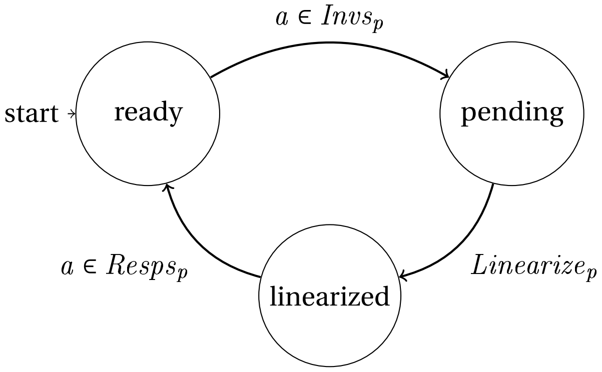

Figure 3.6 – Control flow of a client p in the $Lin$ I/O automaton.

subject to the following conditions.

1. Validity: If a value is decided on, then it must have been previously proposed by a client.
2. Agreement: All clients decide on the same value.
3. Termination: All correct clients eventually decide on a value.

It is relatively easy to see that the traces of the I/O automaton $Lin(Consensus)$ satisfy the validity and agreement properties. Indeed, the "*linearize*" action executes only requests that have been invoked previously, because those requests are the pending request of a client. Thus validity is satisfied. Moreover, in every behavior of the consensus data type, the first executed request determines the output that all subsequent requests will return. Therefore agreement holds. We cannot speak of termination because we consider only finite traces, which do not allow us to define liveness properties.

## 3.4 Refining the Linearizability I/O Automaton

The linearizability I/O automaton, $Lin$, is simple enough to have confidence that it represents our idea of linearizability. However, the experience of the authors has shown that making $Lin$ less deterministic simplifies refining the $Lin$ I/O automaton to prove concrete algorithms correct.

In this section we present the I/O automaton $NDLin$, which is a (more) nondeterministic version of $Lin$. Both have the same set of traces, although we will only show that $NDLin$ implements $Lin$. To obtain the I/O automaton $NDLin$, we will refine the $Lin$ I/O automaton in two steps, obtaining the $Lin'$ I/O automaton in between.

The construction of the $NDLin$ I/O automaton will also show how the idempotence properties of data-type representations are useful.

### 3.4.1 The $Lin'$ I/O Automaton

The $Lin'$ I/O automaton has exactly the same signature as the $Lin$ I/O automaton: its inputs are the invocation actions, its outputs are the response actions and its internal actions are the $Linearize_p$ actions, where $p$ is a client.

The states of the $Lin'$ I/O automaton consists of a $dState$ component and, for every client $p$, of the components $status\,[p]$ and $pending\,[p]$. In contrast to the $Lin$ I/O automaton, there is no $nxtOut\,[p]$ component. Moreover, the status of a client $p$ is now only “ready” or “pending”, and not “linearized”.

As in the $Lin$ I/O automaton, every client is initially ready and $dState$ is initially equal to $\perp$. The value of $pending$ is arbitrary.

An $Inv_p\,(c)$ action is enabled when $p$ is ready. It updates $status\,[p]$ to “pending” and updates $pending\,[p]$ to $\langle p, c\rangle$.

A $Linearize_p$ action is enabled when $p$ is in status “pending”. Its effect is to update the current $\Delta$-state by executing the pending request of $p$, setting $dState$ to $dState \bullet pending\,[p]$. However, in contrast to the $Linearize_p$ transition of the $Lin$ I/O automaton, the ouptut produced by executing the pending request of $p$ is not recorded.

A $Resp_p\,(o)$ action is enabled if the client $p$ is in status “pending”, $dState$ contains the pending request of $p$, and the output $o$ is equal to $\gamma\left(dState, pending\,[p]\right)$. Note that the value of $pending\,[p]$ is left unchanged.

The control flow of a client $p$ in the $Lin'$ I/O automaton is represented graphically in fig. 3.7.

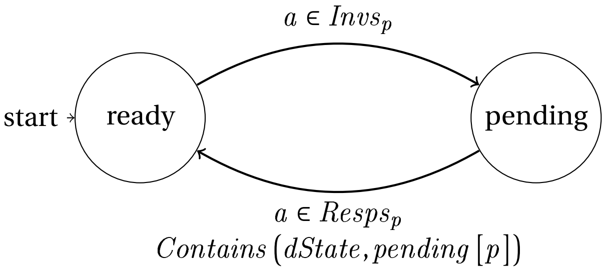

Figure 3.7 – Control flow of a client $p$ in the $Lin'$ I/O automaton.

We see that in order to produce a response to the pending request of a client $p$ it is sufficient that the current $\Delta$-state contains the pending request of $p$. This may happen as a side effect of linearizing the pending request of another client, even if the pending request of $p$ was never linearized. For example, consider the consensus data-type representation presented in section 3.2.3. Suppose that the current state is $\perp$, and that the requests $\langle p_1, v_1\rangle$ and $\langle p_2, v_2\rangle$ are pending. Linearizing the request $\langle p_1, v_1\rangle$ updates the current state to $v_1$. However both $\langle p_1, v_1\rangle$ and $\langle p_2, v_2\rangle$ are contained in $v_1$ because $\perp \star \langle\langle p_1, v_1\rangle\rangle = v_1$ and $\perp \star \langle\langle p_1, v_1\rangle, \langle p_2, v_2\rangle\rangle = v_1$. Therefore, in state $v_1$, the response action of $p_2$ is enabled, even though the $Linearize_{p_2}$ action was never executed.

We also see that the $Lin'$ I/O automaton does not use any $nxtOut\left[p\right]$ component to remember the output that must be returned to the client $p$. Instead, the $Lin'$ I/O automaton returns $\gamma\left(dState, pending\left[p\right]\right)$, even if some other requests were linearized after $p$'s request was linearized.

However, despite its more liberal behavior, the $Lin'$ I/O automaton implements the $Lin$ I/O automaton. The proof shows how this fact relies on the idempotence property of data-type representations.

**Theorem 3.2.** *The $Lin'$ I/O automaton implements the $Lin$ I/O automaton.*

*Proof.* We present a forward simulation $f$ from the I/O automaton $Lin'$ to the I/O automaton $Lin$.

A state $s$ of $Lin'$ is related to a state $t$ of $Lin$ when their $dState$ components are equal and, for every client $p$, the following holds.

1. The client $p$ has the same pending request in $s$ and $t$.
2. (a) If $p$ is "ready" in $s$, then $p$ is also "ready" in $t$ and $nxtOut\left(t\right)\left[p\right]$ is arbitrary;

   (b) if $p$ is in status "pending" in $s$ and $dState\left(s\right)$ contains $pending\left[p\right]$, then $p$ is in status "linearized" in $t$ and $nxtOut\left(t\right)\left[p\right]$ equals $\gamma\left(dState\left(s\right), pending\left[p\right]\right)$;

   (c) if $p$ is in status "pending" in $s$ and $dState\left(s\right)$ does not contain $pending\left[p\right]$, then $p$ is in status "pending" in $t$ and $nxtOut\left(t\right)\left[p\right]$ is arbitrary.

Note that, for every client $p$, unless $p$ is in status "pending" and $dState\left(s\right)$ contains the pending request of $p$, then $nxtOut\left(t\right)\left[p\right]$ is unconstrained.

Let us show that $f$ is forward simulation from $Lin'$ to $Lin$. Assume that $s$ is a reachable state of $Lin'$, that $\left\langle s, a, s'\right\rangle$ is a transition of $Lin'$, and that $t$ is a state of $Lin$ such that $s$ and $t$ are related. Let us show that there exists an execution fragment $e$ whose first state is $t$, whose last state is related to $s'$, and such that

- if $a$ is an external action of the I/O automaton $Lin$, then the trace of $e$ is equal to $\langle a\rangle$;
- if $a$ is not an external action of $Lin$, then the trace of $e$ is the empty sequence.

Remember than when two states are related by $f$, their $dState$ and $pending$ components are equal. We proceed by case analysis on the type of transition that is taken.

1. If $a$ is an invocation action $Invs_p\left(c\right)$, we have two sub-cases:

   (a) Assume that $dState\left(s\right)$ does not contain $p$'s request, $\langle p, c\rangle$. Let $e = \left\langle t, a, t'\right\rangle$ where $t'$ is equal to $t$ except that $pending\left[p\right]$ is updated to $\langle p, c\rangle$ and the status of $p$ is updated to "pending". The state $t'$ is related to the state $s'$ by $f$ and $e$ is an invocation transition of $Lin$, and therefore is an execution fragment of $Lin$.

   (b) Assume that $dState\left(s\right)$ contains $p$'s request already. In this case, the execution $e$ that we are looking for needs to contain an action that linearizes $p$'s request. Let

$e = \langle t, a, t', Linearize_p, t'' \rangle$ where $t'$ is as in the previous case and $t''$ is equal to $t'$ except that $t.nxtOut\,[p]$ is updated to $\gamma(s.dState, \langle p, c\rangle)$ and the status of $p$ is updated to "linearized".

The transition $\langle t, a, t' \rangle$ is an invocation transition of $Lin$.

The transition $\langle t', Linearize_p, t'' \rangle$ appears not to be a "linearize" transition of $Lin$ because we did not update $dState\,(t)$. However, because $dState(t)$ contains the request of $p$, executing the request $\langle p, c\rangle$ on $dState\,(t)$ leaves $dState\,(t)$ unchanged, by the idempotence property of data-type representations (property 1). Therefore $\langle t', Linearize_p, t'' \rangle$ is in fact a "linearize" transition of $Lin$. Therefore $e$ is an execution fragment of $Lin$.

Moreover, $s'$ and $t'$ are related because $s'.dState$ contains $\langle p, c\rangle$, which is consistent with $t'.status\,[p]$ being "linearized".

Therefore we get $e$ is an execution fragment satisfying our goal.

2. Assume that $a$ is a response action $Resp_p\,(o)$. Let $e = \langle t, a, t' \rangle$ where $t'$ is equal to $t$ except that the status of $p$ is updated to "ready".

   Because of the precondition of a $Resp_p\,(o)$ action, we know that $dState\,(s)$ contains $pending\,(s)\,[p]$ and that $p$ is in status "pending". Therefore, by definition of $f$, we have that $nxtOut\,(t)\,[p] = \gamma\left(dState\,(s), \langle p, c\rangle\right)$ and the status of $p$ in $t$ is "linearized". Thus from $t$ to $t'$ the state is updated as in the $Resp_p\left(\gamma\left(dState\,(t), pending\,(t)\,[p]\right)\right)$ transition of $Lin$. Therefore, $\langle t, a, t' \rangle$ is a "response" transition of $Lin$ and $e$ is an execution fragment of $Lin$.

   Moreover, it is easy to see that $s'$ and $t'$ are related, which finishes to prove our goal.

3. Assume that $a$ is a "linearize" action $Linearize_p$ of $Lin'$. Hence, from $s$ to $s'$, $dState$ is updated to $dState\,(s) \bullet pending\,[p]$, resulting in $dState\,(s')$ containing $pending\,[p]$.

   Suppose that $dState\,(s)$ already contains $pending\,[p]$. Then, by the idempotence property of recoverable data-type representations, the action has no effect on the state and the empty execution of initial state $t$, $\langle t\rangle$, satisfies our goal. Therefore we assume that $dState\,(s)$ does not contain $pending\,[p]$.

   Any state $t'$ which is related to $s'$ must have $status\,[p] =$ "commited" and $nxtOut\,[p] = \gamma\left(dState\,(s), pending\,(s)\,[p]\right)$. Thus this must be the case of the last state of the execution $e$ that we are looking for.

   Moreover, there could be a set of clients $Q$, different from $p$, that have a pending request which is not contained in $dState\,(s)$ but which is contained in $dState\,(s')$. Therefore, for every client $q \in Q$, any state $t'$ which is related to $s'$ must have $status\,[q] =$ "commited" and $nxtOut\,[q] = \gamma\left(dState\,(s), pending\,(s)\,[q]\right)$. Thus this must be the case of the last state of the execution $e$ that we are looking for.

   We are therefore going to build an execution $e$ of $Lin$ in which the client $p$ first linearizes its request, followed by all the members of $Q$.

   Let $qs = \langle q_1, \ldots, q_n\rangle$ be a sequence containing at least once (duplicates are allowed) every client of $Q$. Let

$$e = \langle t, Linearize_p, t'_0, Linearize_{q_1}, t'_1, \ldots, Linearize_{q_n}, t'_n \rangle \tag{3.9}$$

where

(a) $t'_0$ is equal to $t$ except that $nxtOut[p]$ is updated to $dState(s) \bullet pending(s)[p]$ and $dState(s') = dState(s) \bullet pending(s)[p]$;

(b) for every $i \in 1..n$, $t'_i$ is equal to $t'_{i-1}$ except that $nxtOut[q_i]$ is updated to $dState(s) \bullet pending(s)[q_i]$.

We see that, for every client $q \in Q \cup \{p\}$, $q$ is in status "linearized" in $t'_n$ and $nxtOut(t'_n)[q] = \gamma(dState(s), pending(s)[p])$. Moreover $dState(t'_n) = dState(s) \bullet pending(s)[p]$. Therefore $s'$ and $t'_n$ are related by the forward simulation relation.

The transition $\langle t, Linearize_p, t'_0 \rangle$ is a $Linearize_p$ transition of $Lin$.

Moreover, for every $i \in 1..n$, $\langle t'_{i-1}, Linearize_{q_i}, t'_i \rangle$ is a $Linearize_{q_i}$ transition of $Lin$, even though we did not update $dState$: by definition of $Q$, we know that $dState(t'_0)$ contains $pending[q_i]$; therefore, by the idempotence property of data-type representations, executing $pending[q_i]$ on $dState(t'_0)$ would leave it unchagned.

Finally, we have shown that $e$ is the execution that we are looking for, and we have proved our goal.

We have covered all the possible types of transitions, therefore the theorem holds. □

Note that we have used a forward simulation and not a refinement mapping. Without adding a history variable to simulate the evolution of the component $nxtOut$, a refinement mapping would not have worked. This is because, for any client $p$, there is no way to reliably determine what $nxtOut[p]$ should be by looking only at $pending[p]$ and $dState$.

### 3.4.2 The $NDLin$ I/O Automaton

We now present the $NDLin$ I/O automaton and show that it refines the $Lin'$ I/O automaton. With the preceding theorem, theorem 3.2, we obtain that $NDLin$ is linearizable.

The $NDLin$ I/O automaton is like the $Lin'$ I/O automaton except that the $Linearize_p$ actions are replace with a single $Linearize$ action, not specific to any client. Otherwise, $NDLin$ has the same external signature, the same set of states, the same initial states, and the same "invocation" and "response" transitions as the $Lin'$ I/O automaton.

The new $Linearize$ transition linearizes multiple requests at once. It is enabled when at least one request is pending. Its effect is to update the current Δ-state by executing *a sequence* $rs$ of pending requests, setting $dState$ to $dState \star rs$. The same effect would be obtained in the $Lin'$ I/O automaton by taking several $Linearize_p$ transitions in a row. Therefore the $NDLin$ I/O automaton refines the $Lin'$ I/O automaton using the identity relation as refinement mapping.

**Theorem 3.3.** *The I/O automaton $NDLin$ implements the I/O automaton $Lin'$.*

*Proof.* The identity relation is a refinement mapping from $NDLin$ to $Lin'$. □

**Corollary 3.1.** *The I/O automaton $NDLin$ implements the I/O automaton $Lin$.*

*Proof.* We have shown that $NDLin \leq Lin'$ (theorem 3.3) and that $Lin' \leq Lin$ (theorem 3.2). Therefore, by transitivity of the implementation relation, we have $NDLin \leq Lin$. □

## 3.5 The Abstraction Theorem

The I/O automaton $SeqImp$, presented below, is a linearizable implementation of $D$ in which the clients take turns for performing their operations: no two operations overlap. The abstraction theorem (theorem 3.4) states that in a system containing a linearizable implementation $Imp$ of $D$, substituting the I/O automaton $SeqImp$ for $Imp$ leaves the set of traces of the system unchanged. Therefore, when reasoning about safety properties of the system, it suffices to examine the system in which $SeqImp$ has been substituted for $Imp$. The substitution simplifies the reasoning problem because, in $SeqImp$, the clients are synchronous instead of asynchronous. Essentially, the abstraction theorem allows one to abstract over the concurrent nature of data-type implementations.

The $SeqImp$ I/O automaton is similar to the $Lin$ I/O automaton: in order to determine the response corresponding to an invocation, it internally queries and updates a copy of the data-type representation $\Delta$. However, unlike the $Lin$ I/O automaton, the $SeqImp$ I/O automaton does not accept any invocation if one invocation is already pending. Therefore its traces are composed of invocation-response pairs which do not overlap.

The I/O automaton $SeqImp$ has the same signature, the same set of states, and the same initial state as the $Lin$ I/O automaton. The $Resp_p(o)$ and $Linearize_p$ transitions of $SeqImp$ are also the same as the ones of $Lin$. The only difference between $Lin$ and $SeqImp$ lies in the $Lin_p(c)$ transition, which has the same effect as in $Lin$ but is enabled only if every client is in status “ready”. Therefore, in every execution of $SeqImp$, there is at most one client which has a pending request.

Let an *application* be an I/O automaton which is compatible with any well-formed implementation of D (see section 3.3.1). Note that such an application takes response actions as input and may output invocation actions.

**Theorem 3.4** (Abstraction Theorem)**.** *If $App$ is an application and $Imp$ is a linearizable implementation of the data type $D$, then the I/O automaton $App \times Imp$ with invocation and responses hidden has exactly the same set of traces as the I/O automaton $App \times SeqImp$ with invocation and responses hidden,*

$$Traces\left(Hide\left(Invs \cup Resps, App \times Imp\right)\right) = Traces\left(Hide\left(Invs \cup Resps, App \times SeqImp\right)\right)$$

Theorem 3.4 casts the result of Filipovic et al. [28] in our framework. We do not give a proof of this theorem and instead refer the reader to Filipovic et al..

## 3.6 The Inter-Object Composition Theorem

Consider two data-type representations $\Delta_1$ and $\Delta_2$ of two data types $D_1$ and $D_2$,

$$\Delta_1 = \langle\langle S_1, C_1, \{\bot_1\}, \delta_1\rangle, O_1, \gamma_1\rangle \qquad \Delta_2 = \langle\langle S_2, C_2, \{\bot_2\}, \delta_2\rangle, O_2, \gamma_2\rangle,$$

such that $C_1 \cap C_2 = O_1 \cap O_2 = \emptyset$.

We define the product of the two data types $D_1$ and $D_2$ as the data type of representation

$$\Delta = \langle\langle S_1 \times S_2, C_1 \cup C_2, \{\langle\bot_1, \bot_2\rangle\}, \delta\rangle, O_1 \cup O_2, \gamma\rangle \tag{3.10}$$

where

1. if $c \in C_1$, then $\langle s_1, s_2\rangle \bullet \langle p, c\rangle = \langle s_1 \bullet \langle p, c\rangle, s_2\rangle$ and $\gamma\left(\langle s_1, s_2\rangle, \langle p, c\rangle\right) = \gamma\left(s_1, \langle p, c\rangle\right)$;
2. if $c \in C_2$, then $\langle s_1, s_2\rangle \bullet \langle p, c\rangle = \langle s_1, s_2 \bullet \langle p, c\rangle\rangle$ and $\gamma\left(\langle s_1, s_2\rangle, \langle p, c\rangle\right) = \gamma\left(s_2, \langle p, c\rangle\right)$.

**Theorem 3.5** (Inter-Object Composition)**.** *Consider two I/O automata $A_1$ and $A_2$. If $A_1$ implements $Lin(\Delta_1)$ and $A_2$ implements $Lin(\Delta_2)$, then the composition of $A_1$ and $A_2$, $A_1 \times A_2$, implements $Lin(\Delta_1 \times \Delta_2)$.*

We do not give a proof of this theorem and instead refer the reader to section 3.1 of Herlihy and Wing [37].

Theorem 3.5 allows us to build an I/O automaton $A$ that is linearizable to a data type $D = D_1 \times D_2$ by composing two I/O automata $A_1$ and $A_2$ which are linearizable to $D_1$ and $D_2$ respectively. Therefore theorem 3.5 is a reduction theorem, in the sense that it allows drawing a conclusion about $A$ by reasoning about a simpler problem, i.e., the linearizability of $A_1$ and $A_2$ when taken in isolation.

## 3.7 The Original Definition of Linearizability

In this section we give the classical, trace-based, definition of linearizability.

### 3.7.1 Happens-before relation

Consider a well-formed trace $t$. We define the relation $<_t$ on the positions of $t$ such that, for all positions $i,j$ in $t$, $i <_t j$ holds when the operation to which $t[i]$ belongs ends before the start of operation to which $t[j]$ belongs .

For example, if

$$t = \left\langle Inv_p(c_1), Res_p(o_1), Inv_q(c_2), Res_q(o_2) \right\rangle, \tag{3.11}$$

then $1 \prec_t 3$ holds because the operation to which $Inv_p(c_1)$ belongs ends with $Res_p(o_1)$ at position 2 and the operation to which $Inv_q(c_2)$ belongs starts with $Inv_q(c_2)$ at position 3. Similarly, we also have $2 \prec_t 3$, $1 \prec_t 4$, and $2 \prec_t 4$:

$$\prec_t = \left\{ \langle 1,3 \rangle, \langle 2,3 \rangle, \langle 1,4 \rangle, \langle 2,4 \rangle \right\} \tag{3.12}$$

However, if

$$t = \left\langle Inv_p(c_1), Inv_q(c_2), Res_p(o_1), Res_q(o_2) \right\rangle, \tag{3.13}$$

then the relation $\prec_t$ is empty.

Formally, if $i, j$ are two positions of $t$, then $i \prec_t j$ holds when there are two positions $i', j'$ such that $i \le i' < j' \le j$, $t[i']$ is a response, $t[j']$ is an invocation, $Proc(t[i]) = Proc(t[i'])$, and $Proc(t[j']) = Proc(t[j])$.

Note that $\prec_t$ is a partial order (i.e. it is reflective, transitive, and antisymmetric). The relation $\prec_t$ is sometimes called the happens-before relation on operations.

### 3.7.2 Safe reordering

Consider another well-formed trace $t'$. We say that $t$ and $t'$ are *weakly equivalent* when for all client $p \in \Pi$, the projection of $t$ onto the actions of $p$ is equal to the projection of $t'$ onto the actions of $p$, $t|p = t'|p$. For example, the following two traces are weakly equivalent.

$$t_1 = \left\langle Inv_p(c_1), Res_p(o_1), Inv_q(c_2), Res_q(o_2) \right\rangle \tag{3.14}$$

$$t_2 = \left\langle Inv_q(c_2), Res_q(o_2), Inv_p(c_1), Res_p(o_1) \right\rangle \tag{3.15}$$

We say that the trace $t'$ is a safe reordering of the trace $t$ when $t$ and $t'$ are weakly equivalent and there exists a bijection $\sigma$ from the positions of $t$ to the positions of $t'$ such that $t[\sigma[i]] = t[i]$ and applying $\sigma$ to $t$ only increases the happens-before relation, $i \prec_t j \Rightarrow \sigma[i] \prec_t \sigma[j]$. For example, the trace $t_1$ is not a safe reordering of $t_2$ but the trace $t_1$ is a safe reordering of the trace

$$t_3 = \left\langle Inv_p(c_1), Inv_q(c_2), Res_p(o_1), Res_q(o_2) \right\rangle. \tag{3.16}$$

However, the trace $t_3$ is not a safe reordering of the trace $t_1$ (the safe reordering relation is not symmetric).

### 3.7.3 Closure of a trace

We now define the *closure* of a trace, which is obtained by removing or completing pending invocations.

The trace $t'$ is a closure of $t$ when, for every client $p$, $t'|p$ ends with a response and either $t'|p$ was obtained by removing the last invocation of $t|p$ (eq. (3.17)), or $t'|p$ was obtained by appending a response action to $t|p$ (eq. (3.18)).

$$\exists a \in \mathit{Invs} : \mathit{Append}\left(t'|p, a\right) = t|p \tag{3.17}$$

$$\exists a \in \mathit{Resps} : t'|p = \mathit{Append}\left(t|p, a\right) \tag{3.18}$$

### 3.7.4 Linearizability

We say that a trace $t$ is linearizable to $D$ when there exists a trace $t_s$ of the sequential implementation of $D$ and a closure $t_c$ of $t$ such that $t_s$ is a safe reordering of $t_c$. In this case we say that $t$ is linearizable to $t_s$ or, equivalently, that $t_s$ is a linearization of $t$.

Note that our definition of linearizability differs slightly from the one usually found in the literature because the traces of the sequential implementation of $D$ contain incomplete actions, i.e., the last action of a client may be an invocation.

Theorem 3.6 asserts that the I/O automaton definition of linearizability coincides with the trace-based definition.

**Theorem 3.6.** *For every data-type representation $\Delta$ of $D$ and for every trace $t$, $t$ is linearizable to $D$ if and only if $t$ is a trace of the I/O automaton $\mathit{Lin}\,(\Delta)$.*

Theorem 3.6 can be seen as a precise formulation of the informal statement saying that "a trace is linearizable if and only if every operation appears to execute atomically at a *linearization point* situated in between its invocation and its response". The linearization points are given by the execution of the $\mathit{Linearie}_p$ actions in the I/O automaton $\mathit{Lin}$.

## 3.8 Conclusion

In this chapter we have defined linearizability to a data type in terms of an I/O automaton based on the notion of data-type representation. We have seen that a data type has different representations which vary in the size of their state space, noting that chosing an appropriate representation may ease a refinement proof of linearizability.

To simplify future refinement proofs, we have refined the Lin I/O automaton to a more nondeterministic version called $\mathit{NDLin}$. We have seen that the idempotence property of data-type representations play a crucial role in the correctness of $\mathit{NDLin}$.

We have presented two well-known reduction theorems that simplify linearizability proofs: the inter-object composition theorem and the abstraction theorem. Finally, we have also seen the equivalent, original, trace-based specification of linearizability.

In the next chapters, we will see that another form of reduction property is needed to simplify our understanding of *robust* linearizable algorithms.

# 4 Adaptive Algorithms and Modular Reasoning

## 4.1 Introduction

In this chapter we define *adaptive algorithms*, which model *robust* distributed systems, and we define what it means to reason *modularly* about an adaptive algorithm and why it is desirable.

Adaptive algorithms model distributed and linearizable data type implementations that have several *modes* of execution, that *dynamically change mode* in response to the changes of behavior of their environment, and whose modes are *encapsulated* so as to minimize the dependencies between two modes.

An adaptive behavior is a requirement for a robust system: In practice, the environment of a distributed system changes unpredictably, and most existing algorithms only exhibit good performance in particular conditions. Therefore, to be robust, i.e., maintain high performance in all scenarios, a system must dynamically adapt its strategy.

Using *adaptive algorithms*, as we define them in this chapter, is one way to achieve dynamic adaptation to a changing environment. Adaptive algorithms are composed of a set of modes (or sub-algorithms), they choose the best mode available for the current operating conditions, and they constantly re-evaluate their choice in order to match the changes of their environment.

We have seen in the introduction that building adaptive algorithms ad-hoc is not practical. When changing mode, a linearizable adaptive algorithm must obviously preserve linearizability, thus modes need to synchronize on a mode change. Therefore, to allow arbitrary changes of modes, one must make sure that any mode can synchronize properly with any other. If each mode uses its own ad-hoc conventions for synchronization, checking that all modes can synchronize properly implies to examine $O\left(n^2\right)$ cases, where $n$ is the number of modes. Second, incremental design is unpractical. If one wants to incrementally design an adaptive algorithm constituted of $n$ modes, then on is faced in the worst case with a number of cases to consider of $\sum_{i=1}^{n} i^2 = O\left(n^3\right)$: if adding a new modes causes changes to the existing modes, one has to check anew that all the modes are compatible with each other. Clearly, such a situation is not practical.

To simplify the development of adaptive algorithms, we first require that their different modes be encapsulated in an interface that minimizes the dependencies between modes. This interface consists of a unique entry point and a unique exit point per client. Appart from the calls to this interface, there is no communication between different modes. It may seem strange to put the inter-mode interface on the clients because mode changes should be transparent to the clients. However, localizing the inter-mode interface on some other components of the system would require making assumptions about the internal components of the modes. We rule out this possibility in order not to restrict unnecessarily the possible mode implementations. Moreover, in practice, a thin interface could easily hide mode changes from client applications and, to guarantee smooth mode changes, the role of client can be played by some servers belonging to the service provider.

Instead of synchronizing modes through ad-hoc conventions, we propose to build adaptive algorithms around *modular properties*. A modular property $P$ is a correctness condition which applies to a mode taken in isolation and such that if all the modes of an adaptive algorithm $A$ individually satisfy $P$, then $A$ is linearizable to $D$.

Observe that if every mode of an adaptive algorithm $A$ satisfies the modular property $P$, then any new mode satisfying $P$ may be added to $A$ without changing the existing modes. Moreover, in order to prove that the new mode satisfies $P$, one does not need to know anything about the other existing modes. Modular properties thus solve the scalability problem that ad-hoc approaches suffer from.

## 4.2 Related Work

The idea of improving the robustness and performance of distributed systems through adaptation is quite old has a rich literature literature.

Pedone [81] shows through several examples how optimistic distributed protocols can boost the performance of distributed systems.

Hiltunen and Schlichting [38] presents an informal model for adaptive fault-tolerant systems and proposes to build adaptive algorithms by composing event-driven micro-protocols, giving a few examples. At a high level, their modeling approach is similar to ours, but they do not discuss the practical problem of reasoning about adaptive systems.

Chang et al. [15] observes that high performance in fault tolerant algorithms requires adaptation. They propose a method, similar to speculation, for avoiding the overhead of full-fledged fault tolerance when it is not necessary. They propose building algorithms out of modes that are specialized for particular fault patterns. They apply their ideas to an atomic broadcast protocol, studying in depth the performance of the modes scheduling policy. They eschew the issue of maintaining the properties of atomic broadcast when switching mode by allowing disorderly delivery of messages during mode changes.

Later works emphasizes the issue of coordination of adaptation. Renesse et al. [83] and Oreizy et al. [78] study adaptive algorithms that briefly stop servicing requests in order to change mode. Bickford et al. [7] rigorously model and analyze adaptive distributed algorithms (called Hybrid Protocols in their work) which can change mode without synchronization. Their work is formalized in the NUPRL [20, 3] proof assistant.

Chen et al. [17] propose a general model for adaptive systems and an implementation using the Cactus system. They implement and evaluate an adaptive group communication protocol that continues servicing requests while changing mode. Wojciechowski, Rütti, and Schiper [85, 93, 86] covered the issue of Dynamic Protocol Update, with a focus on the problem of synchronizing updates of group communication protocol. They also present ways of changing group communication algorithm without stopping the system while maintaining the properties of group communication.

McKinley et al. [68] and Oreizy et al. [77] survey the literature on adaptive software.

Devising a scheduling policy, i.e. an algorithm to choose when to trigger adaptation and which mode to switch to, is orthogonal to our work. However it is an issue that is also covered by the literature, for example in the works of Rosa et al. [84]

A more general problem than the one of building adaptive algorithms is to formally model systems in which components can be created or removed dynamically. Bozga et al. [11] propose Dy-BIP, an extension of the BIP framework [6] that supports dynamic addition and removal of components and interactions between components. Attie and Lynch [5] propose a similar extension to the I/O automata framework.

## 4.3 Modeling Adaptive Algorithms with I/O Automata

We would like to model, using I/O automata, systems that are composed of a set of *modes* and which run as follows.

At a high level, the system first choses an initial mode, *instantiates* it, and runs it. The initial mode may *abort* at any time; when it does so, a new mode is chosen, instantiated, and run in place of the previous mode. This process can repeat any number of times. Moreover, the system also has a scheduling policy, i.e., an algorithm used to choose when to abort and which mode to run next.

At a lower level, a client runs only one mode at a time and can enter a *mode instance* only once. This one call used to enter a mode instance, modeled by a *switch action*, forms the interface that encapsulates mode instances. Moreover, we let the clients change mode asynchronously from each other.

Modeling adaptive algorithms with I/O automata poses two problems: first, the theory of I/O automata does not support the dynamic creation of components, and, second, the policy governing the dynamic selection of modes may depend on complex runtime properties that are difficult to model (like the throughput of the algorithm, the average latency, etc.).

We avoid the two problems be abstracting over the dynamic nature of the changes of modes and over the scheduling policy. We will see that our abstraction of the dynamic nature of changes is sound, i.e., it is an over-approximation of the behavior of the adaptive algorithm. However, we leave the problem of the soundness of the abstraction over the scheduling policy to the user who wishes to use our framework. She must make sure that her model of her adaptive algorithm soundly models reality.

## 4.4 A Model for Adaptive Algorithms

Remember that we consider a fixed data type $D$.

We define an adaptive algorithm as set of *modes*, each mode representing a particular algorithm. A mode is a function from natural numbers to I/O automata called *mode instances*. If $M$ is a mode, then we say that the I/O automaton $M[i]$ is the $i^{th}$ mode instance of $M$. Moreover, we say that an I/O automaton $A$ is an $i^{th}$ *mode instance* when there exists a mode $M$ of the adaptive algorithm where $A = M[i]$.

We now assume that all the actions $a$ that we consider have an *instance number*, denoted $Num(a)$, usually appearing as superscript in action names. For example, an invocation action of instance number $i$ is denoted $Inv_p^i(c)$, and $Num\left(Inv_p^i(c)\right) = i$.

For a sequence of I/O automata to qualify as a mode, its instances need to be *well-formed*, a concept that we now define.

### 4.4.1 Well-Formed Mode Instances

Let $V$ be a set whose members we call *switch values*.

When $i > 1$, the $i^{th}$ instance of a mode is well-formed when its traces $t$ are such that, for every client $p$, the projection $t|p$ starts with an action of the form $Switch_p^i(c, v)$, for a command $c$ and a switch value $v$, then continues by alternating response actions, of the form $Resp_p^i(o)$, and invocation actions, of the form $Inv_p^i(c)$, until a pending request of $p$ is aborted by a $Switch_p^{i+1}(c, v)$ action.

A $Switch_p^i(c, v)$ action models the client $p$ entering an $i^{th}$ mode instance after its request $\langle p, c\rangle$ was aborted in in the mode instance numbered $i-1$. Conversely, an action $Switch_p^{i+1}$ models the client $p$ switching to the next mode instance, numbered $i+1$, because the $i^{th}$ mode instance aborted its request. When discussing the $i^{th}$ instance of a mode, we say that actions of the form $Switch_p^i(c,v)$ are *init actions* and that the actions of the form $Switch_p^{i+1}$ are *abort actions*. Moreover, given a trace $t$ of an $i^{th}$ instance, the switch values appearing in the init actions found in $t$ are called *init values*, and the switch values appearing in the abort actions found in $t$ are called *abort values*. Note that the abort action of an $i^{th}$ mode instance are the init actions of an $i+1^{th}$ mode instance.

When $i = 1$, the $i^{th}$ instance is the first mode instance. There is no previous mode instance that can switch to the first mode instance. Therefore, a first mode instance is well-formed when its traces $t$ are such that, for every client $p$, the projection $t|p$ starts with an invocation action, of the form $Inv_p^1(c)$, then continues by alternating response actions, of the form $Resp_p^1(o)$, and invocation actions, of the form $Inv_p^1(c)$, until a pending request of $p$ is aborted by a $Switch_p^2(c,v)$ abort action.

We define $Switch^i$ as the set of all the init actions of an $i^{th}$ mode instance,

$$Switchs^i = \bigcup_{p\in\Pi, c\in C, v\in V} Switch_p^i(c,v), \tag{4.1}$$

and we define $Switch_p^i$ as the set of all the init actions of the client $p$ in an $i^{th}$ instance,

$$Switchs_p^i = \bigcup_{c\in C, v\in V} Switch_p^i(c,v). \tag{4.2}$$

We define $Invs^i$, $Invs_p^i$, $Resps^i$, and $Resps_p^i$ similarly,

$$Invs^i = \bigcup_{p\in\Pi, c\in C} Inv_p^i(c) \qquad Invs_p^i = \bigcup_{c\in C} Inv_p^i(c) \tag{4.3}$$

$$Resps^i = \bigcup_{p\in\Pi, o\in O} Resp_p^i(o) \qquad Resps_p^i = \bigcup_{o\in O} Resp_p^i(o). \tag{4.4}$$

To compose consecutive mode instances, we require that, for every $i \in \mathbb{N}$ and for every modes $M$ and $N$, a well-formed $i^{th}$ mode instance $M[i]$ and a well-formed $(i+1)^{th}$ mode instance $N[i+1]$ be compatible and that the switch actions $Switchs^{i+1}$ be outputs of $M[i]$ and inputs of $N[i+1]$.

In section 3.3.1, we have defined the I/O automaton *Seq* to formalize well-formed data-type implementations. In the following paragraphs, we define the I/O automaton $ModeInst(i)$ to formalize the concept of well-formed mode instances.

The I/O automaton $ModeInst(i)$ is obtained as the composition, for every client $p$, of the I/O automata $ModeInst\left(i,p\right)$,

$$ModeInst(i) = \prod_{p\in\Pi} ModeInst\left(i,p\right). \tag{4.5}$$

The inputs of $ModeInst\left(i,p\right)$ are the invocation actions of process $p$, $Invs_p^i$, and, if $i > 1$, the init actions of process $p$, $Switchs_p^i$. The outputs of $ModeInst\left(i,p\right)$ are the abort actions of process $p$, $Switchs_p^{i+1}$, and the response actions of process $p$, $Resps_p^i$.

A state of the I/O automaton $ModeInst\left(i,p\right)$ describes the status of the client $p$, which can be either "idle", "ready", "pending", or "aborted". If $i > 1$, then every client is initially "idle". Otherwise, when $i = 1$, every client is initially "ready".

The transition relation of $ModeInst\left(i,p\right)$ implements the behavior described above.

1. An init action $Switch_p^i\left(c,v\right)$ is enabled when the client $p$ is in status "idle" (possible only if $i = 1$). Its effect is to set the status of the client to "pending".
2. A response action $Resp_p^i\left(o\right)$ is enabled when $p$ is in status "pending". Its effect is to set the status of $p$ to "ready".
3. An invocation action $Inv_p^i\left(c\right)$ is enable when $p$ is ready. Its effect is to set the status of $p$ to "pending".
4. An abort action $Switch_p^{i+1}\left(c,v\right)$ is enabled when $p$ is in status "pending" and the pending request of $p$ is $\langle p,c\rangle$. It sets the status of $p$ to "aborted". Once $p$ has aborted, the execution of $ModeInst\left(i,p\right)$ stops.

The transition relation of $ModeInst\left(i,p\right)$ is represented graphically in fig. 4.1, for $i > 1$, and in fig. 4.2, for $i = 1$.

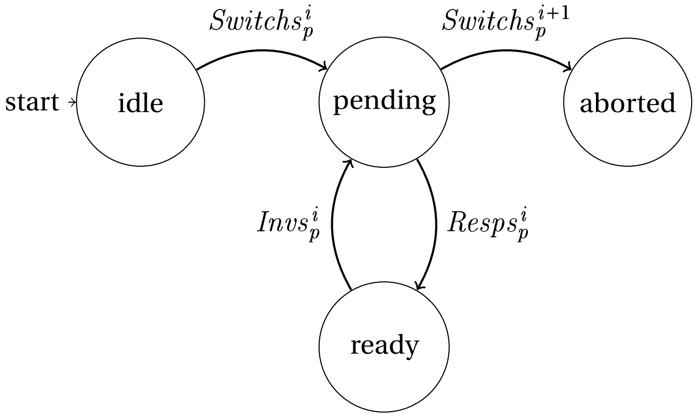

Figure 4.1 – The transition relation of $ModeInst\left(i,p\right)$, when $i > 1$.

Note that if $p \neq q$ them $ModeInst\left(i,p\right)$ and $ModeInst\left(i,q\right)$ have no common action. Thus, in $ModeInst\left(i\right)$, the two components $ModeInst\left(i,p\right)$ and $ModeInst\left(i,q\right)$ execute completely asynchronously. Notably, processes can change mode asynchronously.

Given a trace $t$ of $ModeInst\left(i\right)$, we say that $v \in V$ is an *init value* if $v$ appears as argument of a switch action of instance number $i$ and we say that $v$ is an *abort value* if $v$ appears as argument of a switch value of instance number $i+1$.

Finally, a well-formed mode instances is defined as an I/O automaton that implements $ModeInst\left(i\right)$ for some $i \in \mathbb{N}$ and whose internal actions all have the instance number $i$. The

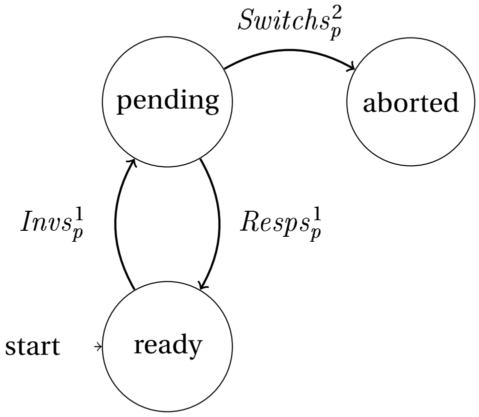

Figure 4.2 – The transition relation of $ModeInst\left(1,p\right)$.

requirement on the instance number of internal actions ensures that, when $i \neq j$, an $i^{th}$ mode instance and a $j^{th}$ mode instance are compatible I/O automata.

### 4.4.2 Composing Modes Instances

By definition of the I/O automaton $ModeInst\,(i)$, if $M$ and $N$ are two modes, then, for any two natural numbers $i$ and $j$,

1. if $i \neq j$, then the mode instances $M\,[i]$ and $N\left[j\right]$ are compatible I/O automata;
2. if $|j - i| > 1$, then $M\,[i]$ and $N\left[j\right]$ have no common actions;
3. if $j = i + 1$, then a process that aborts in $M\,[i]$ starts its execution in $N\left[j\right]$, accurately modeling switching from one mode instance to the next.

The property stated in item 1 above implies that mode instances of different index can be composed. Moreover, the properties of items 2 and 3 imply that only consecutive mode instances may communicate, and that information flows only from the instance of smallest index to the instance of largest index. This communication between consecutive mode instances models processes running the smallest mode instance aborting and changing to the next mode instance.

Finally, note that if one composes a set of instances containing one instance of index $i$ for every natural number $i$, then, hiding the switch actions, one obtains a well-formed data-type implementation.

#### Example: the I/O Automaton $ModeInst\,(1) \times ModeInst\,(2)$

The interface of a well-formed mode instance and the restriction on its traces allows one to compose two consecutive mode instances to obtain an I/O automaton representing an adaptive algorithm that executes the first instance and then switches to the second instance, as shown in the following example.

Consider the I/O automaton $A = \mathit{ModeInst}\,(1) \times \mathit{ModeInst}\,(2)$. By definition of $\mathit{ModeInst}\,(i)$ we have that

$$A = \left(\prod_{p\,\in\,\Pi} \mathit{ModeInst}\left(1,p\right)\right) \times \left(\prod_{p\,\in\,\Pi} \mathit{ModeInst}\left(2,p\right)\right). \tag{4.6}$$

Applying lemma 2.1, we obtain

$$\mathit{Traces}\,(A) = \mathit{Traces}\Big(\prod_{p\,\in\,\Pi}\Big(\mathit{ModeInst}\left(1,p\right) \times \mathit{ModeInst}\Big(2,p)\Big)\Big)\Big). \tag{4.7}$$

For every client $p$, a state of the I/O automaton $\mathit{ModeInst}\left(1,p\right) \times \mathit{ModeInst}\left(2,p\right)$ is a pair whose first element is the status of $p$ in the first mode instance and whose second element is the status of $p$ in the second mode instance. In the initial state, every client $p$ is in status “ready” in the first mode instance and in status “idle” in the second. The transition relation of the composition of the two instance is represented graphically in fig. 4.3.

Note that a process starts by emitting an invoke action of instance number 1, followed by a sequence of response and invoke actions alternating in lockstep, all with instance number 1, until the process emits a switch action with instance number 2, which is followed by a sequence of response and invoke actions alternating in lockstep, all with instance number 2, until the process emits a switch action of instance number 3. This sequence of actions models a process starting its execution in a mode instance of index 1 and at some point switching to a mode instance of index 2, which terminates when trying to switch to a mode instance of index 3 because there is no such instance in the system.

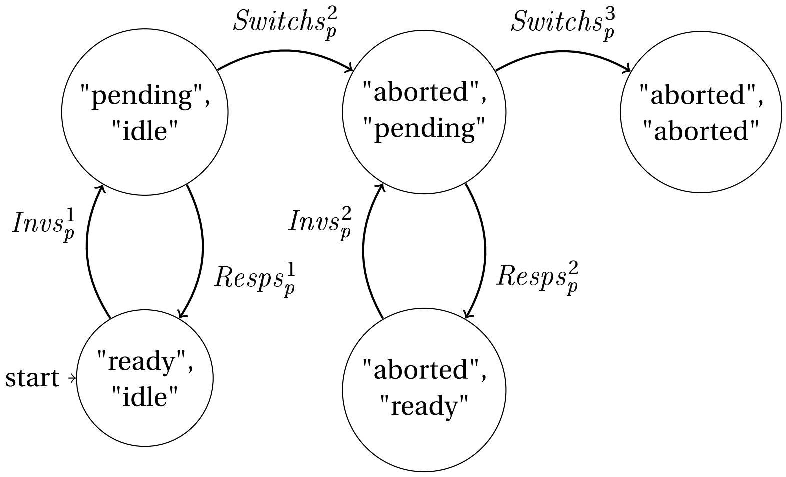

Figure 4.3 – The transition relation of $\mathit{ModeInst}\left(1,p\right) \times \mathit{ModeInst}\left(2,p\right)$ where unreachable states have been removed.

**Example: Compositing Three Mode Instances**

Figure 4.4 represents graphically how the interfaces of mode instances compose. The figure represents a system consisting of three modes instances $M_1[1], M_2[2], M_3[3]$, two processes p and q and a client application using the interface of the data type $D$.

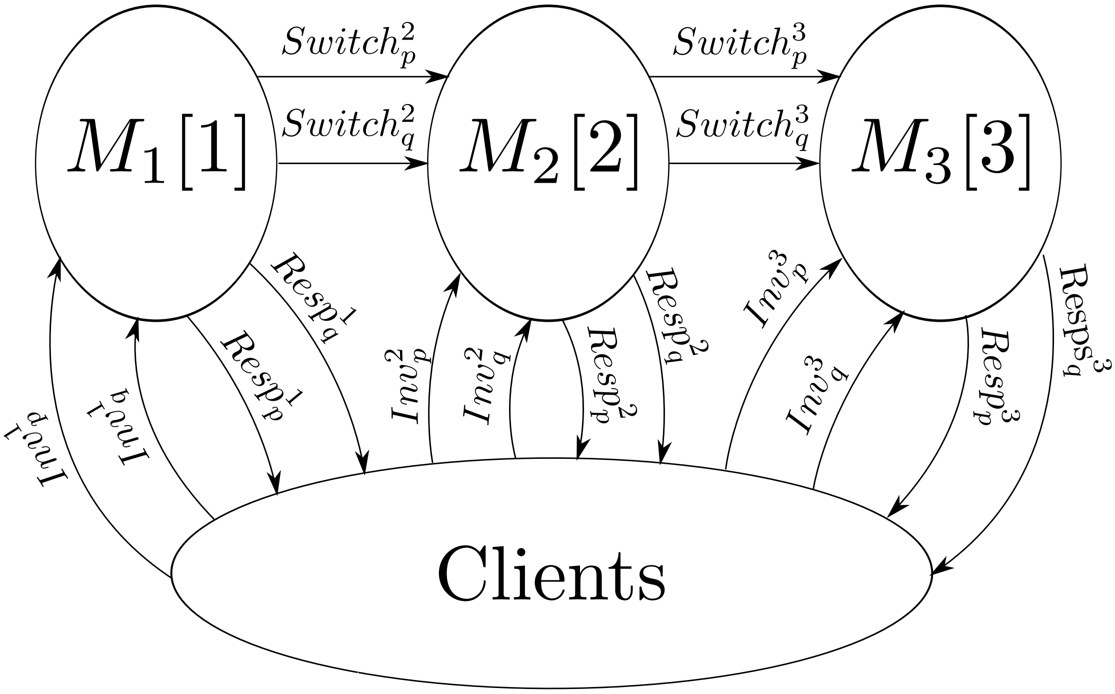

Figure 4.4 – Interfaces in a system composed of a three mode instances (of three different modes $M_1$, $M_2$, and $M_3$), of two processes p and q, and of a client application.

### 4.4.3 A Correctness Condition for Adaptive Algorithms

We have defined above an adaptive algorithm as a set of modes. Then we have defined modes, mode instances, and we have seen that mode instances can be composed. However, we have not seen exactly how these definition relate to our idea of a real adaptive algorithm. Notably, we have avoided mentioning the problems related to the dynamic nature of an adaptive algorithm and to the scheduling policy. We now address those concerns and, in consequence, define what it means for an adaptive algorithm to be correct.

First note that the interface of a mode instance does not contain any actions that could model a scheduling policy component to indicate to the clients when to change mode and which mode to switch to. Thus the scheduling policy is not part of our model, and it is the responsibility of our user to make sure that this does not make her model unsound. In the algorithm that we present in later chapters, clients can change mode instance at any time, nondeterministically.

We now define a correctness condition for adaptive algorithms and we show that it soundly abstracts over the dynamic nature the scheduling of modes.

We define the *mode schedules* of an adaptive algorithm $A$ as the I/O automata $Sched$ such that there exists a sequence $\langle M_1, \ldots, M_n \rangle \in A^*$ of modes such that $Sched$ is the product, for every position $i$ in the sequence, of the $i^{th}$ instance of the mode $M_i$,

$$Sched = \prod_{i \in 1..n} M_i[i]. \tag{4.8}$$

We say that the adaptive algorithm $A$ is correct when every mode schedule of $A$, with switch actions hidden, is a linearizable implementation of $D$. Therefore, in a correct adaptive algorithm, the asynchronous changes of mode can be made transparent to an application, which only accesses the implementation through invocation and response actions, by hiding instance numbers and forwarding invocation to the instance that is currently active on the given client.

Now consider a real adaptive algorithm modeled by the set of modes $A$. An execution of a mode schedule $Sched$ of $A$ corresponds to a run of the adaptive algorithm in which mode instances are scheduled according to their order in the sequence $Sched$. Moreover, for any possible succession of modes observed in a run of a real adaptive algorithm, there is a corresponding mode schedule of $A$ in which the modes appear in the same order as in the run. Therefore, if $A$ is correct, then any run of the real algorithm (where modes are scheduled dynamically) is linearizable. Conversely, of the real algorithm is correct, the $A$ is correct. Note that as explained above, we leave the burden of soundly abstracting the interaction of the mode instances with the scheduling policy to our user and we assume that her abstraction is sound.

By definition of a mode instance, two consecutive mode instances in a mode schedule must synchronize using the init values received (one per process), because the init values are the only information transfered from one mode instance to the next. This restriction simplifies reasoning about adaptive algorithms, as we will see in the next section.

## 4.5 Modular Properties

Our definition of the correctness of an adaptive algorithm requires that any mode schedule be linearizable. Checking that every mode schedule is linearizable one by one is of course not feasible because there are infinitely many mode schedules. A more realistic approach would consist in showing that for any two modes $M_1$ and $M_2$ of A, switching from an instance of $M_1$ to an instance of $M_2$ preserves linearizability. However this approach suffers from the scalability problem and the incremental design problem identified in the introduction: There a $n^2$ mode changes to consider, n being the number of modes of A, and adding a new mode to an existing algorithm, as would be done when designing an algorithm incrementally, may require in the worst case to reconsider all the $n^2$ previous cases and $n+1$ new cases. To solve these problems, we propose a third approach: using modular properties.

A modular property reduces the correctness of an adaptive algorithm to the correctness of each of its modes, when taken independently of the others. This statement is formalized in the *modularity theorem* below (theorem 4.1). With the abstraction theorem (theorem 3.4) and the inter-object composition theorem (theorem 3.5), the modularity theorem constitutes a third reduction theorem that simplifies the analysis of linearizable adaptive algorithms

Define $Invs^{i,j}$ as the set of all the invocation actions whose instance number is comprised between $i$ and $j$ with $i$ and $j$ included,

$$Invs^{i,j} = \bigcup_{k \in i..j} Invs^k \tag{4.9}$$

Define $Resps^{i,j}$ and $Switchs^{i,j}$ similarly,

$$Resps^{i,j} = \bigcup_{k \in i..j} Resps^k; \qquad Switchs^{i,j} = \bigcup_{k \in i..j} Switchs^k. \tag{4.10}$$

Define $\pi_{i,j}(A)$ as the I/O automaton obtained by hiding in the I/O automaton $A$ the switch actions whose instance number lies between $i+1$ and $j-1$ with bounds included,

$$\pi_{i,j}(A) = hide\left(A, Switchs^{i+1,j-1}\right). \tag{4.11}$$

Also remember that $\pi_{i/r}(A)$ is the projection of $A$ onto the invocation and response actions,

$$\pi_{i/r}(A) = proj\left(A, Invs \cup Resps\right). \tag{4.12}$$

Let $P$ be a two-dimensional array of I/O automata, $P\left[i,j\right]$ where $i,j \in \mathbb{N}$. We say that $P$ is *modular* when $P$ is well-formed, linearizable, and idempotent:

1. $P$ is *well-formed:* for every $i \in \mathbb{N}$, $P[i,i+1]$ is a well-formed $i^{th}$ mode instance and the I/O automata $P[1,i]$ and $P[i,i+1]$ are compatible;

2. $P$ is *linearizable:* for every $i \in \mathbb{N}$, $P[1,i]$ is linearizable to $D$;

3. $P$ is *idempotent:* for every natural number $i > 1$, the composition of $P[1,i]$ and $P[i,i+1]$, with the intermediate switch actions hidden, implements $P[1,i+1]$,

$$\pi_{1,i+1}(P[1,i] \times P[i,i+1]) \le P[1,i+1]. \tag{4.13}$$

We say that an adaptive algorithm $A$ satisfies a modular property $P$ when for every mode $M \in A$ and for every natural number $i$, the $i^{th}$ mode instance of $M$ implements $P[i,i+1]$:

$$\forall M \in A, i \in \mathbb{N} : M[i] \le P[i,i+1]. \tag{4.14}$$

### 4.5.1 The Modularity Theorem

**Theorem 4.1** (Modularity Theorem)**.** *If $P$ is modular and $A$ satisfies $P$, then $A$ is correct.*

Informally, forgetting about the compatibility of signatures, the proof of theorem 4.1 has the following structure. First, we show by induction on the length $n$ of a sequence of modes $Ms$ that $Ms$ implements $P[1, n+1]$. In the inductive step we prove that $Ms_{n+1} = Append(Ms_n, M)$ implements $P[1, n+2]$ using the inductive hypothesis ($Ms_n$ implements $P[1, n+1]$), the fact that $M[n+1]$ implements $P[n+1, n+2]$ (because $A$ satisfies the modular property $P$), and the idempotence property of modular properties ($P[1, n+1] \times P[n+1, n+2]$ implements $P[1, n+2]$). Second, with the linearizability property of modular properties, we get from $Ms \le P[1, n+1]$ that $Ms$ is linearizable.

The proof of the modularity theorem is conceptually simple but requires carefully manipulating the signatures of the different I/O automata. We first need a few lemmas.

**Lemma 4.1.** *If $P$ is modular and $i > 1$, then*

$$Inputs(P[1,i]) = Invs^{1,i-1}, \tag{4.15}$$

$$Outputs(P[1,i]) = Resps^{1,i-1} \cup Switchs^{2,i}, \tag{4.16}$$

$$Inputs(P[i,i+1]) = Switchs^{i} \cup Invs^{i}, \tag{4.17}$$

$$Outputs(P[i,i+1]) = Resps^{i} \cup Switchs^{i+1}, \tag{4.18}$$

*Proof.* Follows from the fact that $P$ is well-formed and idempotent. □

The following corollary of lemma 4.1 will be useful in proving theorem 4.1:

**Corollary 4.1.**

$$\forall i,j \in \mathbb{N} : \left(\pi_{i/r} \circ \pi_{i,j}\right)\left(P\left[i,j\right]\right) = \pi_{i/r}\left(P\left[i,j\right]\right), \tag{4.19}$$

*Proof.* By lemma 4.1 □

**Lemma 4.2.** *If $Ms$ is a sequence of modes of an adaptive algorithm $A$ and $n = Len(Ms)$, then*

$$\pi_{1,n+1}\left(\prod_{i \in 1..n} Ms[i][i]\right) = \pi_{1,n+1}\left(\pi_{1,n}\left(\prod_{i \in 1..(n-1)} Ms[i][i]\right) \times Ms[n]\right) \tag{4.20}$$

Let us now prove the modularity theorem.

**Theorem 4.1** (Modularity Theorem)**.** *If $P$ is modular and $A$ satisfies $P$, then $A$ is correct.*

*Proof.* By the definition of the correctness of an adaptive algorithms , we must show that for every mode schedule $Sched$ of $A$, $\pi_{i/r}(Sched)$ is linearizable. Expanding the definition of a mode schedule, we must prove that:

$$\forall Ms \in A^* : \pi_{i/r}\left(\prod_{i \in Dom(Ms)} Ms[i][i]\right) \leq \text{Lin}(\Delta) \tag{4.21}$$

We proceed by induction on the length of the sequence $Ms$. Note that we will often implicitly use the monotonicity of the composition and projection operators with respect to the implementation relation (theorems 2.1 and 2.3), as well as lemma 4.1.

Let $n = Len(Ms)$, the length of Ms. Define the inductive property, $IP(Ms)$, as follows.

$$IP(Ms) = \pi_{1,n+1}\left(\prod_{i \in 1..n} Ms[i][i]\right) \leq P[1, n+1] \tag{4.22}$$

Suppose that we prove that $IP(Ms)$ holds for every mode sequence $Ms$. Then we have

$$\pi_{i/r}\left(\pi_{i,n+1}\left(\prod_{i \in 1..n} Ms[i][i]\right)\right) \leq \pi_{i/r}(P[1, n+1]). \tag{4.23}$$

Therefore, by corollary 4.1,

$$\pi_{i/r}\left(\prod_{i \in 1..n} Ms[i][i]\right) \leq \pi_{i/r}(P[1, n+1]) \tag{4.24}$$

Moreover, because $P$ is linearizable , we have $\pi_{i/r}(P[1, n+1]) \leq Lin(\Delta)$, which proves the theorem. Therefore, establishing that $IP$ holds for all $Ms \in A^*$ would prove our goal.

Let us now prove by induction that IP holds for all sequences of modes.

1. If $Ms = \langle\rangle$ then we are done because the empty I/O automaton implements any I/O automaton.
2. If $Ms = \langle M_1 \rangle$ then

$$\pi_{1,2}\left(\prod_{i \in 1..n} Ms[i][i]\right) = M_1[1]. \tag{4.25}$$

   Since $A$ satisfies $P$ and $M_1$ is a mode of $A$, we have that the first instance of $M_1$, $M_1[1]$, implements $P[1,2]$. Therefore, by transitivity of $\leq$ and monotonicity of projection, we get $IP(Ms)$.
3. Now let us show the inductive step. Suppose that the sequence of modes $Ms$ is obtained by appending a mode $M$ of $A$ to the sequence of modes $Ms'$. Suppose that $IP(Ms')$, the induction hypothesis, holds. Let $n$ be the length of $Ms'$.

By lemma 4.2,

$$\pi_{1,n+2}\left(\prod_{i\,\in\,1..(n+1)} Ms[i][i]\right) \leq \pi_{1,n+2}\left(\pi_{1,n+1}\left(\prod_{i\,\in\,1..n} Ms'[i][i]\right) \times M[n+1]\right). \tag{4.26}$$

Moreover, by the induction hypothesis,

$$\pi_{1,n+1}\left(\prod_{i\,\in\,1..n} Ms'[i][i]\right) \leq P[1,n+1]. \tag{4.27}$$

Therefore,

$$\pi_{1,n+2}\left(\prod_{i\,\in\,1..(n+1)} Ms[i][i]\right) \leq \pi_{1,n+2}\left(P[1,n+1] \times M[n+1]\right). \tag{4.28}$$

Since $M \in A$ and $A$ satisfies $P$ (eq. (4.14)), we get

$$\pi_{1,n+2}\left(\prod_{i\,\in\,1..(n+1)} Ms[i][i]\right) \leq \pi_{1,n+2}\left(P[1,n+1] \times P[n+1,n+2]\right). \tag{4.29}$$

Finally, with the idempotence property of $P$ (eq. (4.13)), we conclude that

$$\pi_{1,n+2}\left(\prod_{i\,\in\,1..(n+1)} Ms[i][i]\right) \leq P[1,n+2]. \tag{4.30}$$

□

## 4.6 Conclusion

In this chapter we have proposed a formal model of adaptive distributed algorithms, represented as sets of modes. A mode represents a particular strategy available to the adaptive algorithm.

Modes can be instantiated one after the other to form a chain called mode schedule. A mode schedule represents all the runs of an adaptive algorithm in which the modes are scheduled in the particular order in which they are instantiated. An adaptive algorithm is said correct when all its mode schedules are linearizable. With the notion of mode schedule we avoid introducing the dynamic creation of components in our model.

Our model soundly abstracts over the dynamic nature of the scheduling of modes. However, it is the responsibility of the user who wishes to use our framework to soundly abstract the components responsible for the scheduling policy.

Using our model, we have defined the notion of modular property. A modular property is a correctness condition that applies to a mode in isolation and guarantees that a set of modes individually satisfying the property can be composed without modifications to form a correct adaptive algorithm.

Building an adaptive algorithm around a modular property would solve that scalability problem of the design process and make it incremental. However, it remains to show whether modular properties exist that can be applied in practice exist.

In the next chapter we present a modular property that is both general, applying to any data type, and efficiently implementable.

# 5 Speculative Linearizability

## 5.1 Introduction

In the preceding chapter, we have motivated the need for modular reasoning and we have precisely defined modular properties, which enable scalable and incremental design of adaptive algorithms. However one important question remain: are there modular properties which are efficiently implementable in the shared-memory or message-passing models of computation?

In this chapter we propose a modular property called *speculative linearizability*. Speculative linearizability takes a parameter that allows one to instantiate it for any given data type. In the next two chapters, we show that speculative linearizability can be efficiently implemented in the message-passing model and we present a proof-of-concept implementation in shared memory.

$SLin$ is a modular property, a two-dimensional array of I/O automata, where the $SLin\,(\Delta)\,[i, i+1]$ I/O automaton models an $i^{th}$ mode instance which behaves *speculatively*, i.e., which only responds to invocations under optimistic assumptions. If the optimistic assumptions hold, the optimistic mode instance performs very efficiently because it does not waste resources preparing for worst-case conditions. However, if the optimistic assumptions do not hold, the state of the system can become inconsistent. In this case, the clients must detect the inconsistency, *abort* their execution of the current mode instance and *switch* to the next mode instance, passing a Δ-state as switch value. When the clients abort, the task of recovering a consistent state and continuing the execution is picked up by the next mode instance. To recover a consistent state, the next mode instance uses the Δ-states received as switch values from the previous instance. The array of of I/O automata $SLin\,(\Delta)$ formally specifies this process and, notably, defines how the execution of a mode should be encoded in the switch values in order for the next mode to continue the execution and ensure that it remains linearizable.

The parameter Δ of the family of I/O automata $SLin\,(\Delta)$ must be a *recoverable data-type representation*, abbreviated *RDR*, which is a special case of data-type representation. An RDR guarantees that a consistent state can be recovered from a set of different states of the RDR. The notion of RDR is based on the notion of C-Struct Set proposed in [49] to generalize the Consensus problem.

## 5.2 Related Work

Several reduction theorems can simplify the analysis of adaptive distributed algorithms. In the next three paragraphs we reference reduction theorems that apply to distributed algorithms in general. The Abstract framework provides, to our knowledge, the only reduction theorem specifically targeting adaptive algorithms.

The abstraction and compositional properties of Linearizability [37, 51, 52, 28], presented in chapter 3, are useful in simplifying the development of distributed systems. To reason about the safety of a distributed system containing linearizable objects, it suffices to consider only the executions in which the linearizable objects are accessed sequentially, thus abstracting over concurrent accesses of the objects. Moreover, accessing two linearizable objects in parallel, without any synchronization, results in an execution which is linearizable to a simple product of the two base objects. This property reduces the task of building a linearizable implementation of a composite data type to the task of building linearizable implementations of each of the components of the data type.

Elrad and Francez [24] define communication-closed layers and show that to reason about the safety of algorithms composed of communication-closed layers, one can assume that the layers are sequentially composed, without interleaving. Charron-Bost and Schiper [16] build on this work to propose a model unifying the treatment of process faults and communication faults in distributed algorithms that evolve in communication-closed rounds. Their work is not directly applicable to our case because algorithms which continuously receive requests, as opposed to one-shot algorithms like consensus, cannot be decomposed in communication-closed layers: their clients can always interact across layers.

Cut-off theorems are another kind of reduction theorems: they reduce the correctness of a system to the correctness of its instances that have a fixed, usually small, size. For example, some properties of networks of processes connected in a ring have cutoff sizes below 5 [27], meaning that verifying them on a system containing 5 processes is sufficient to conclude that the system is correct for any number of processes. Emerson and Kahlon [26] derives cutoff bounds for systems whose processes are instances of a generic process template. Examples include a cache coherence protocol. A later paper [25] generalizes the method to networks of heterogeneous processes.

The Abstract framework [35] proposes a reduction theorem, called the Composition Theorem, that is the main inspiration behind the Speculative Linearizability framework. Roughly speaking, the Abstract correctness properties define a modular property that applies to the $Generic$ data type defined in section 3.2.3. The Composition Theorem proves that the modular property is idempotent. In the Abstract framework, adaptive algorithms do not optimize the execution of commuting requests and must maintain full execution histories in their data-structures. Inspired by work on the Generalized Consensus problem [49], we have in turn generalized the Abstract framework to allow optimized execution of commuting requests and to minimize the size of the data-structures that implementations must use.

## 5.3 Recoverable Data-Type Representations (RDRs)

Remember that we consider a data-type representation $\Delta = \langle \Sigma, O, \gamma \rangle$ of $D$, where $\Sigma = \langle S, C, \{\bot\}, \delta \rangle$ is state machine. The states $s \in S$ of the state machine are called $\Delta$-states. To define recoverable data-type representations, we need the concepts of ordering of $\Delta$-state and of greatest lower bound.

We say that a $\Delta$-state $d$ is smaller than a $\Delta$-state $d'$, denoted $d \preceq d'$, when there exists a sequence of requests $rs$ such that executing $rs$ starting from $d$ results in $d'$,

$$d \preceq d' \Leftrightarrow \exists rs : d' = d \star rs. \tag{5.1}$$

Note that the “smaller than” relation on $\Delta$-states is not necessarily a partial order, for example when the transition relation $\delta$ has cycles.

A $\Delta$-state $d$ is a *lower bound* of a set of $\Delta$-states $ds$ when $d$ is smaller than every member of $ds$. We write $GLB\,(ds)$ for the *greatest lower bound*, or glb for short, of the $\Delta$-states $ds$, when it exists. Also note that the glb of a set of $\Delta$-states does not necessarily exist.

We say that $\Delta$ is a recoverable data-type representation when the following three properties hold:

**Property 3** (Antisymmetry)**.** *The “smaller than” relation on $\Delta$-states, $\preceq$, is antisymmetric.*

**Property 4** (Existence of GLB)**.** *Every two $\Delta$-states have a unique greatest lower bound.*

**Property 5** (Consistency)**.** *If $d1 \preceq d2 \preceq d3$ and $d3 = d1 \star rs$, where $d_1$, $d_2$, and $d_3$ are three $\Delta$-states and $rs$ is a sequence of requests, then*

1. *there exists $rs'$ such that $d2 = d1 \star rs'$ and every request of $rs'$ is a request of $rs$ and*
2. *there exists $rs''$ such that $d3 = d2 \star rs''$ and every request of $rs''$ is a request of $rs$.*

Properties 3 and 4 imply that that the set $S$ of $\Delta$-states and the “smaller than” relation form a *meet semi lattice* with $\bot$ as least element: by definition, $\preceq$ is reflexive and transitive; with property 3, we get that $\preceq$ is a partial order; with property 4 we have that $\langle S, \preceq \rangle$ is a meet semi-lattice.

We will see in the proof of the idempotence of speculative linearizability (section 5.4.4) that properties 3 to 5 are crucial for the successful recovery of an aborted instance of $SLin$.

The reader who is familiar with the Generalized Consensus problem [49] will recognize the similarity between RDRs and C-Struct Sets. Although similar, RDRs have a notion of behavior that includes the outputs that clients receive, whereas C-Struct Sets do not.

The Isabelle/HOL formalization of Recoverable Data-Type Representations appears in appendix B, and its TLA+ counterpart in appendix A.

We now show that any data type has a RDR and, in particular, we present the *History RDR*, $H^{\#}(D)$, of a data type. Like $\mathit{Fold}(\Delta)$, which is a minimal data-type representation, $H^{\#}(D)$ is a minimal *recoverable* data-type representation.

**Lemma 5.1.** *Every data type $D$ has a recoverable data-type representation.*

*Proof.* $\mathit{Unfold}(\Delta)$ is a recoverable data-type representation of $D$. □

The state of the representation $\mathit{Unfold}(\Delta)$, defined in section 3.2.4, is the full sequence of requests that have been executed so far, modulo duplicated requests. In this case, a $\Delta$-state $d$ is smaller than a $\Delta$-state $d'$ if $d$ is a prefix of $d'$. Moreover, the greatest lower bound of $d$ and $d'$ is their longest common prefix.

The RDR $\mathit{Unfold}(\Delta)$ is not a very efficient representation because it uses full execution histories. In section 3.2.4 we have seen that $\mathit{Fold}(\Delta)$ minimizes the number of states that a representation can have. However, $\mathit{Fold}(\Delta)$ is not always a RDR because it may introduce cycles in the state transition graph representing $\delta$.

In order to obtain RDRs with small state spaces, we now introduce the History RDR $H^{\#}(D)$, where # is a *dependency relation* of $D$.

### 5.3.1 The History Data-Type Representation

We say that two requests $r$ and $r'$ commute when, for every behavior $b = \langle op_1, \ldots, op_n\rangle$ of $D$, if $r$ and $r'$ appear in two adjacent operations $op_i$ and $op_{i+1}$, then the behavior obtained by swapping $op_i$ and $op_{i+1}$ is also a behavior of $D$. Note that this means that we can swap commuting requests without affecting subsequent requests and without changing the output that the two swapped requests receive. The commutativity property of requests is formalized in a *dependency relation* which contains every pair of requests that do not commute.

It is often difficult to determine whether two requests commute. Instead, we can use an over-approximation of the *dependency relation* by including requests that commute in the dependency relation. We say that a relation # over requests is a dependency relation of $D$ when # is symmetric and, if $r$ and $r'$ are two requests that do *not* commute, then $\langle r, r'\rangle \in \#$. Note that a dependency relation is necessarily symmetric. When $\langle r, r'\rangle \in \#$ we say that $r$ and $r'$ are (mutually) dependent.

Given a dependency relation #, we say that two sequences of requests $rs$ and $rs'$ are *equivalent* when one can be obtained from the other by applying a permutation that preserves the relative order of dependent requests. More precisely, the sequences of requests $rs$ and $rs'$ are equivalent when there exists a permutation $\sigma$ such that, for every position $i$, $rs\,[i] = rs'\,[\sigma\,[i]]$ and, for every position $j$, if $i < j$ and $\langle rs\,[i]\,, rs\,[j]\rangle \in \#$, then the permutation $\sigma$ preserves the order of $i$ and $j$, $\sigma\,[i] < \sigma\,[j]$.

Note that if $rs$ and $rs'$ are equivalent, then, for every request $r$,

$$\gamma\,(\bot \star rs, r) = \gamma\,(\bot \star rs', r)\,, \tag{5.2}$$

and

$$Eq\,(Append\,(rs, r)) = Eq\,(Append\,(rs', r)) \tag{5.3}$$

Moreover, the equivalence relation on sequence of requests is symmetric: a dependency relation is symmetric, therefore, considering $\sigma^{-1}$, the inverse of the permutation $\sigma$, we get that $rs'$ and $rs$ are equivalent.

The equivalence relation is symmetric, transitive, and reflexive, therefore we can define the equivalence class $Eq\,(rs)$ of a sequence of requests $rs$ and we know that the equivalence classes form a partition of the set of sequences of requests. We now consider a dependency relation #.

We now define the *history data-type representation*, $H^{\#}\,(D)$. The states of $H^{\#}\,(D)$ are the equivalence classes of the dependency relation #. The transition function $\delta^{\#}$ maps the equivalence class $Eq\,(rs)$ of a sequence of requests $rs$ and a new request $r$ to the equivalence class of the concatenation of $rs$ and $r$,

$$\delta_{\#}\,(Eq\,(rs)\,, r) = Eq\,(Append\,(rs, r))\,. \tag{5.4}$$

Equation (5.4) properly defines $\delta_{\#}$ because if $rs'$ is equivalent to $rs$, then, by eq. (5.3),

$$Eq\,(Append\,(rs', r)) = Eq\,(Append\,(rs, r))\,. \tag{5.5}$$

Moreover, we define the output function $\gamma_{\#}$ such that the output obtained by executing a request $r$ on the equivalence class $Eq\,(rs)$ is equal to the output obtained by executing in $\Delta$ the request $r$ on the $\Delta$-state $\bot \star rs$,

$$\gamma_{\#}\,(Eq\,(rs)\,, r) = \gamma\,(\bot \star rs, r)\,. \tag{5.6}$$

Equation (5.4) properly defines $\delta_{\#}$ because if $rs'$ is equivalent to $rs$, then, by eq. (5.2),

$$\gamma\,(\bot \star rs, r) = \gamma\,(\bot \star rs', r)\,. \tag{5.7}$$

Now define the history data-type representation $H^{\#}(D)$ as the data-type representation whose states are the equivalence classes of #, whose initial state is the equivalence class of the empty sequence of requests, whose transition function is $\delta_{\#}$, and whose output function is $\gamma_{\#}$,

$$H^{\#}(D) = \langle\langle H, \{Eq(\langle\rangle)\}, C, \delta_{\#}\rangle, O, \gamma_{\#}\rangle. \tag{5.8}$$

We now have the following important property.

**Theorem 5.1.** *If # is a dependency relation of $D$ then the data-type representation $H^{\#}(D)$ is a* recoverable *data-type representation.*

*Proof.* See section 4.4 of Lamport [49], where the properties of interest are proved in the context of C-Struct Sets. The proof of [49] is based on ideas from trace theory presented in Mazurkiewicz [67]. □

Theorem 5.1 is important because, in contrast to $Unfold(\Delta)$, executing commutative requests in any order always leads to the same $\Delta$-state in $H^{\#}(D)$. With the $unfold(\Delta)$ RDR, executing commutative requests in different orders lead to different $\Delta$-states. We will see in chapter 6 that this property allows algorithms to execute commutative requests without synchronization.

## 5.4 Speculative Linearizability

Speculative Linearizability is a modular property, $SLin[i,j]$, which uses a set of switch values $V = S$, where $S$ is the set of states of a Recoverable Data-Type Representation. In this chapter we precisely define the I/O-automata family $SLin[i,j]$. The Isabelle/HOL formalization of Speculative Linearizability appears in appendix B, and its TLA+ counterpart in appendix A.

For every $i \in \mathbb{N}$, the $SLin[i,i+1]$ I/O automaton is a well-formed $i^{th}$ mode instance. This means that, when $i > 1$, clients start their execution with an init action, followed by a response, then an invocation, then a response, etc. until they abort a pending request by emitting an abort action. If $i = 1$, then the clients start their execution with an invocation action instead of an init action.

We first examine the I/O automaton $SLin[1,i]$ where $i > 1$.

### 5.4.1 The I/O Automaton $SLin[1,i]$

The definition of the $SLin[1,i]$ I/O automaton ensures that, as required of a modular property, $SLin[1,i]$ is linearizable when its abort actions are hidden and $SLin[1,2]$ is a well-formed first mode instance.

### Signature

As noted above, every client starts its execution with an invocation action, therefore the $SLin\,[1,i]$ I/O automaton has no input switch actions. The input actions of $SLin\,[1,i]$ are the invocation actions whose instance number belongs to $1..(i-1)$,

$$Inputs\,(SLin\,[1,i]) = Invs^{1,i-1}. \tag{5.9}$$

The set of output actions of the I/O automaton $SLin\,[1,i]$ consists of the response actions whose instance number belongs to $1..(i-1)$ and of the switch actions whose instance number is $i$,

$$Outputs\,(SLin\,[1,i]) = Resps^{1,i-1} \cup Switchs^{i}. \tag{5.10}$$

The signature of $SLin\,[1,i]$ contains all invocations and responses in the instance number range $1..(i-1)$ because $SLin$ needs to satisfy the idempotence property of modular properties. This will become clear once we define, in the next section, the I/O automaton $SLin\,[i,j]$ in the general case where $i,j \in \mathbb{N}$.

The $SLin\,[1,i]$ I/O automaton is very similar to the $NDLin$ I/O automaton of section 3.4 except that it has abort actions. Like in the $NDLin$ I/O automaton, the internal actions of the I/O automaton $SLin\,[1,i]$, of the form $Linearize^{1}$, are actions which linearize a whole sequence of pending requests at once.

### State Space and Transition Relation

The state of $SLin\,[1,i]$ consists of four components:

1. $dState$, tracking the current state of the RDR $\Delta$;
2. $abortVals$, tracking the set of abort values that have been produced so far;
3. $status\,[p]$, tracking the control flow location of $p$, for every client $p$;
4. $pending\,[p]$, containing the pending request of $p$, for every client $p$.

We define $PendingReqs$ as the set of requests $r$ such that there exists a process $p$ in status "pending" or "aborted" such that $pending\,[p] = r$,

$$PendingReqs = \{pending\,[p] : status\,[p] \in \{\text{"pending"}, \text{"aborted"}\}\}. \tag{5.11}$$

Initially, $dState$ is $\perp$, $abortVals$ is the empty set, and, for every client $p$, $status\,[p] = \text{"ready"}$ and $pending\,[p]$ is arbitrary. As in the $ModeInst\,(1,p)$ I/O automaton, a client $p$ can be either in status "ready", "pending", or "aborted".

Given a state of $SLin\,[1,i]$, we say that $d$ is a *choosable-Δ-state*, $d \in Choosable$, when

1. there is a sequence of pending requests $rs \in Seq\,(PendingReqs)$ where $d = dState \star rs$ and

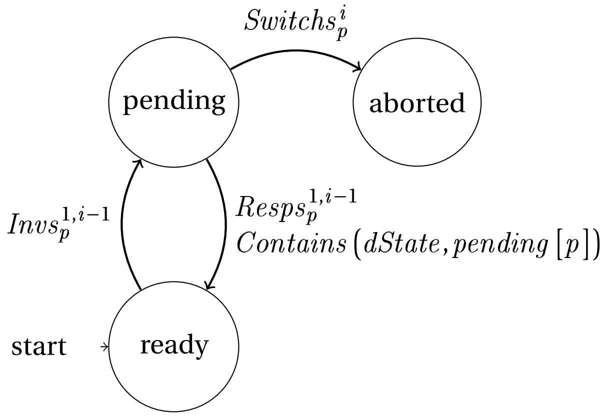

Figure 5.1 – The control flow of a process p in the $SLin\left[1,i\right]$ I/O automaton

2. $d$ is bounded above by every member of $abortVals$.

We will see below that the $Linearize^1$ action updates $dState$ to a choosable-Δ-state.

We now describe the transition relation of $SLin\left[1,i\right]$.

1. The invocation action $Inv_p^m\left(c\right)$ where $m \in 1..(i-1)$ is enabled when $p$ is ready. Its effect is to update $pending\left[p\right]$ to $\langle p, c\rangle$ and to set $status\left[p\right]$ to "pending". The client $p$ now has a pending request. Note that this action is the same as the $Inv_p\left(c\right)$ action of the $NDLin$ I/O automaton.
2. The $Linearize^1$ action is similar to the $Linearize$ action of the $NDLin$ I/O automaton, linearizing multiple pending requests at once, but it restricts the possible new values of $dState$ to the ones that are bounded above by every abort value: The action $Linearize^1$ is enabled when at least one client is in status “pending” and there is at least one choosable Δ-state. Its effect is update $dState$ to a choosable Δ-state.
3. The response action $Resp_p^m\left(o\right)$ where $m \in 1..(i-1)$ is enabled when $p$ is in status “pending”, $dState$ contains the pending request of $p$, and the output $o$ is equal to the output obtained by executing the pending request of $p$ on $dState$, $o = \gamma\left(dState, pending\left[p\right]\right)$.
4. The abort action $Switch_p^i\left(c, av\right)$ is enabled when $p$ is in status “pending”, the pending request of $p$ is $\langle p, c\rangle$, and the abort value $av$ is of the form $av = dState \star rs$ where $rs$ is a sequence of pending requests. The effect of the abort action is to update $dState$ to $dState \cup \{av\}$. The abort action models the client $p$ extracting an “approximate” but safe estimate of $dState$ from an implementation that has been corrupted by overly optimistic speculative updates.

The control flow of a client $p$ is represented graphically in fig. 5.1.

### An Important Invariant

**Invariant 1.** *In every reachable state of $SLin\left[1,i\right]$, every abort value $av \in abortVals$ is of the form $dState \star rs$, where $rs \in Seq\left(PendingReqs\right)$.*

As we will see in the next subsection, in the composition $SLin[1,i] \times SLin\left[i,j\right]$, the I/O automaton $SLin\left[i,j\right]$ relies on the invariant to recover a consistent state of the RDR Δ and continue the execution where $SLin[1,i]$ left it, preserving linearizability.

### 5.4.2 Linearizability of $SLin$

We see that, ignoring the abort actions, the actions of the $SLin[1,i]$ I/O automaton are all actions of the $NDLin$ I/O automaton. Moreover, the abort action only stops a client, setting its status to "aborted". Therefore it is easy to show that $SLin[1,i]$ implements $NDLin$.

**Theorem 5.2.** *For every $i \in \mathbb{N}$, the projection of $SLin[1,i]$ onto the invocation and response actions implements the I/O automaton $NDLin$.*

*Proof.* Let $f$ be the function mapping a state of $s$ of $SLin[1,i]$ to a state $t$ of $NDLin$ such that

1. the $dState$ and $pending$ components of $s$ and $t$ are equal;
2. the status of a client $p$ in $t$ is the same as the status of $p$ in $s$ except that if $status(s)\left[p\right]$ = "aborted", then $status(t)\left[p\right]$ = "pending".

It is easy to see that the function $f$ is a refinement mapping from $SLin[1,i]$ to $NDLin$. □

**Corollary 5.1** (Linearizability of $SLin$). *For every $n \in \mathbb{N}$, the projection of $SLin[1,i]$ onto the invocation and response actions is linearizable.*

*Proof.* Using corollary 3.1 ($NDLin \leq Lin$) and the transitivity of the implementation relation. □

### 5.4.3 The I/O Automaton $SLin\left[i,j\right]$

For $SLin$ to be a modular property, the composition $SLin[1,i] \times SLin[i,i+1]$, for $1 < i$, must implement $SLin[1,i+1]$. Therefore, the I/O automaton $SLin[i,i+1]$ must be able to continue the execution started by $SLin[1,i]$ while preserving linearizability. Moreover, $SLin[i,i+1]$ must be a well-formed mode instance. We will now define $SLin\left[i,j\right]$ with these constraints in mind. In fact, $SLin$ has the more general property $SLin[1,i] \times SLin\left[i,j\right] \leq SLin\left[1,j\right]$, for $1 < i < j$.

#### Signature

The input actions of $SLin\left[i,j\right]$ are the invocation actions whose instance number belongs to $i..(j-1)$ and the switch actions of instance number $i$ (the init actions),

$$Inputs(SLin[1,i]) = Invs^{i,j-1} \cup Switchs^{i}. \tag{5.12}$$

The set of output actions of the I/O automaton $SLin\left[i,j\right]$ consists of the response actions whose instance number belongs to $i..(j-1)$ and of the switch actions whose instance number is $j$ (the abort actions),

$$Outputs\left(SLin\left[i,j\right]\right) = Resps^{i,j-1} \cup Switchs^{j}. \tag{5.13}$$

The internal actions of $SLin\left[i,j\right]$ are the actions of the form $Linearize^{i}$ and $Recover^{i}$.

We see that $SLin\left[i,i+1\right]$ has the signature of a well-formed mode instance, that the signature of $SLin\left[1,i\right]$ is compatible with the signature of $SLin\left[i,j\right]$, and that the external signature of $\pi_{1,j}\left(SLin\left[1,i\right] \times SLin\left[i,j\right]\right)$ is equal to the external signature of $SLin\left[1,j\right]$, as required of a well-formed modular property.

### State Space

The state of $SLin\left[i,j\right]$ consists of 6 components:

1. $dState$, tracking the current $\Delta$-state;
2. $initVals$, tracking the set of init values that have been received so far;
3. $abortVals$, tracking the set of abort values that have been produced so far;
4. $initialized$, a boolean;
5. $status\left[p\right]$, tracking the control flow location of $p$, for every client $p$;
6. $pending\left[p\right]$, containing the pending request of $p$, for every client $p$.

We see that a state of $SLin\left[i,j\right]$ has all the components of a state of $SLin\left[1,i\right]$ plus the boolean $initialized$ and the set of $\Delta$-states $initVals$. We will see that when $initialized$ is true, $SLin\left[i,j\right]$ executes exactly like $SLin\left[1,i\right]$.

Initially, $dState$ is $\perp$, the sets $initVals$ and $abortVals$ are empty, $initialized$ is false, and, for every client $p$, $status\left[p\right] =$ "idle" and $pending\left[p\right]$ is arbitrary.

As in the $ModeInst\left(i,p\right)$ I/O automaton, a client $p$ can be either in status “idle”, “ready”, “pending”, or “aborted”. Note that, in contrast to $SLin\left[1,i\right]$, the initial status of a client is “idle”, not “ready”.

### Transition Relation

Given a state $s$ of $SLin\left[i,j\right]$, we define four sets of $\Delta$-states: the set of glbs of init values, $G$, the set of *safe init values*, $SafeInits$, the set of *choosable values*, $Choosable$, and the set of *safe abort values*, $SafeAborts$. We will see that safe init values are used in the $Recover^{i}$ action to initialize $dState$, choosable values are used in the $Linearize^{i}$ action to update $dState$, and safe abort values are used in the $Switch_{p}^{j}$ actions as abort values.

The main intuition behind the definitions is that, in a state $\langle s_1, s_2\rangle$ of the composition $SLin\left[1,i\right] \times SLin\left[i,j\right]$, the glb of any subset of $initVals\left(s_2\right)$ is of the form $dState\left(s_1\right) \star rs$

where $rs \in PendingReqs\,(s_1)$. The $Recover^i$ action uses this property to simulate a $Linearize^1$ action. Also see the sketch of the proof of the idempotence property of $SLin$, which depends heavily on the definitions of $G$, $SafeInits$, $Choosable$, and $SafeAborts$.

Let $G$ be the set of the $\Delta$-states $g$ where $g$ is the glb of a nonempty subset $initVals$,

$$G = \left\{GLB\,(ivs) : ivs \subseteq initVals\right\}. \tag{5.14}$$

We say that a $\Delta$-state $d$ is a ***safe init value***, $d \in SafeInits$, when

1. $d$ is of the form $g \star rs$ where $g \in G$ and $rs \in Seq\,(PendingReqs)$ is a sequence of pending requests and
2. $d$ is bounded above by every member of $abortVals$,

$$\begin{aligned} SafeInits = \{d \in S : \exists g \in G, rs \in Seq\,(PendingReqs) :\\ d = g \star rs \wedge \forall av \in abortVals : d \preceq av\} \end{aligned} \tag{5.15}$$

We say that a $\Delta$-state $d$ is a ***choosable*** $\Delta$***-state***, $d \in Choosable$, when

1. $d$ is greater than or equal to $dState$ and
2. $d$ is bounded above by every member of $abortVals$ and
3. there is a sequence of pending requests $rs$ where either
   (a) $d = dState \star rs$ or
   (b) there exists $g \in G$ such that $d = g \star rs$.

More formally, the set of safe $\Delta$-states is defined as follows.

$$\begin{aligned} SafeDStates = \{d \in S : dState \preceq d \wedge (\forall av \in abortVals : d \preceq av)\\ \wedge \exists rs \in Seq\,(PendingReqs) : d = dState \star rs \vee \exists g \in G : d = g \star rs\}. \end{aligned} \tag{5.16}$$

We now define the set of ***safe abort values***, $SafeAborts$.

1. If the boolean $initialized$ is false, then the safe abort values are the $\Delta$-states of the form $g \star rs$ where $g \in G$ and $rs \in Seq\,(PendingReqs)$ is a sequence of pending requests.
2. If $initialized$ is true, then the safe abort values are the $\Delta$-states $d$ such that
   (a) $d$ is greater than or equal to $dState$ and
   (b) there is a sequence of pending requests $rs$ where either
      i. $d = dState \star rs$ or
      ii. there exists $g \in G$ such that $d = g \star rs$.

Formally, if $initialized$ is false, then

$$SafeAborts = \{g \star rs : g \in G \wedge rs \in Seq\,(PendingReqs)\}, \tag{5.17}$$

and if *initialized* is true, then

$$\begin{aligned}&SafeAborts = \{d \in S : dState \preceq d \\ &\quad \wedge \exists rs \in Seq\left(PendingReqs\right) : d = dState \star rs \vee \exists g \in G : d = g \star rs\}\end{aligned} \tag{5.18}$$

We now describe the transition relation of $SLin\left[i,j\right]$.

1. The init action $Switch_p^i\left(c, iv\right)$ is enabled when $p$ is in status “idle”. Its effect is to update $pending\left[p\right]$ to $\langle p, c\rangle$, to add $iv$ to the set $initVals$, and to set $status\left[p\right]$ to "pending".
2. the $Recover^i$ action is enabled when the boolean $initialized$ is false and the set $initVals$ is nonempty. Its effect is to set $dState$ to a safe init value $iv \in SafeInits$ and to set $initialized$ to true. Note that if $initVals$ is nonempty, then $SafeInits$ is nonempty too.
3. The invocation action $Inv_p^m\left(c\right)$ where $m \in i..(j-1)$ is enabled when $p$ is ready. Its effect is to update $pending\left[p\right]$ to $\langle p, c\rangle$ and to set $status\left[p\right]$ to "pending".
4. The $Linearize^i$ action is enabled when at least one client has a pending request and the boolean $initialized$ is true. Its effect is to linearize an arbitrary sequence of pending requests by updating $dState$ to a choosable Δ-state $d \in Choosable$.
5. The response action $Resp_p^m\left(o\right)$ where $m \in i..(j-1)$ is enabled when $p$ is in status “pending”, the boolean $initialized$ is true, $dState$ contains the pending request of $p$, and the output $o$ is equal to the output obtained by executing the pending request of $p$ on $dState$, $o = \gamma\left(dState, pending\left[p\right]\right)$. The effect of the response action is to update the status of $p$ to “ready”.
6. The abort action $Switch_p^j\left(c, av\right)$ is enabled when $p$ is in status “pending”, the pending request of $p$ is $\langle p, c\rangle$, and $av$ is a safe abort value $av \in SafeAborts$.

The control flow of a client $p$ is represented graphically in fig. 5.1.

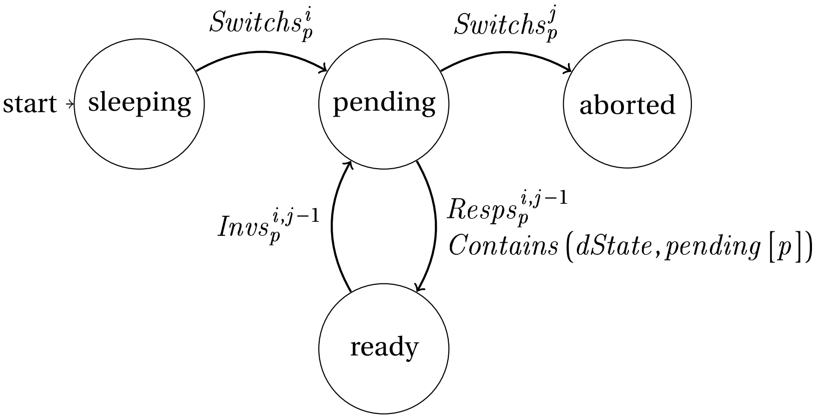

Figure 5.2 – The control flow of a process p in the $SLin\left[i,j\right]$ I/O automaton when $1 < i < j$.

### 5.4.4 Idempotence of $SLin$

We have shown in section 5.4.2 that $SLin$ is linearizable. To prove that $SLin$ is a modular property, we still need to show that $SLin$ is idempotent and well-formed. We now address idempotence. The invariants and refinement proof sketch below should help the reader understand the definitions of the previous subsection. A mechanical proof of the idempotence theorem, in Isabelle/HOL, appears in appendix B.

**Theorem 5.3** (Idempotence of $SLin$)**.** *The array of I/O automata $SLin$ is idempotent.*

To sketch the proof of this result we first need to establish a few invariants of the I/O automaton $SLin\,[1,i] \times SLin\,[i,i+1]$.

**Lemma 5.2.** *Consider three $\Delta$-states $d_0$, $d_1$, and $d_2$, a set of requests $R$, and two sequences of requests $rs_1, rs_2 \in R^*$. If $d_1 = d_0 \star rs_1$ and $d_2 = d_0 \star rs_2$, then there exists a sequence of requests $rs \in R^*$ such that $GLB\,(d_1, d_2) = d_0 \star rs$.*

*Proof.* Follows from the consistency property of RDRs. □

Consider a state $\langle s_1, s_2\rangle$ of $SLin\,[1,i] \times SLin\,[i,i+1]$.

Let $PendingReqs'$ be the set of requests $r$ which are pending in $s_2$ or such that there exists $p$ where $status\,(s_1)\,[p] =$ "pending" and $pending\,[p] = r$,

$$PendingReqs' = PendingReqs\,(s_2) \cup \{pending\,[p] : status\,(s_1)\,[p] = \text{"pending"}\} \qquad (5.19)$$

**Invariant 2.** *If $initialized\,(s_2)$ is false, then $PendingReqs'$ is a equal to $PendingReqs\,(s_1)$.*

**Invariant 3.** *If $initialized\,(s_2)$ is false, then for every safe init value $siv \in SafeInits\,(s_2)$, there exists a sequence of pending requests $rs \in PendingReqs'$ such that $siv = dState\,(s_1) \star rs$.*

**Invariant 4.** *If $initialized\,(s_2)$ is false, then for every safe abort value $sav \in SafeAborts\,(s_2)$, there exists a sequence of pending requests $rs \in PendingReqs'$ such that $sav = dState\,(s_1) \star rs$.*

**Invariant 5.** *If $initialized\,(s_2)$ is true and $av \in SafeAborts\,(s_2)$, then there exists a sequence of requests $rs' \in Seq\,(PendingReqs')$ such that $av = dState\,(s_2) \star rs'$.*

**Invariant 6.** *If $initialized\,(s_2)$ is true and $d \in Choosable\,(s_2)$, then there exists a sequence of requests $rs' \in Seq\,(PendingReqs')$ such that $d = dState\,(s_2) \star rs'$.*

Invariants 3 and 4 follow from the conjunction of invariant 1, presented in the previous section, lemma 5.2, and invariant 2. Invariants 5 and 6 follow from the conjunction of invariant 1, presented in the previous section, lemma 5.2, and the definition of $PendingReqs'$.

Let us now sketch the proof of theorem 5.3

**Theorem 5.3** (Idempotence of $SLin$). *The array of I/O automata $SLin$ is idempotent.*

*Proof.* Define the function $f$ mapping a state $\langle s_1, s_2\rangle$ of $SLin\,[1,i] \times SLin\,[i,i+1]$ to the state $s$ of $SLin\,[1,i+1]$ where

1. the boolean $initialized\,(s)$ is true;
2. if $dState\,(s_2) = \bot$, then $dState\,(s)$ is equal to $dState\,(s_2)$, else $dState\,(s)$ is equal to $dState\,(s_1)$;
3. for every client $p$, if $status\,(s_1)\,[p] =$ "aborted", then $status\,(s)\,[p] = status\,(s_2)\,[p]$, else $status\,(s)\,[p] = status\,(s_1)\,[p]$;
4. for every client $p$, if $status\,(s_1)\,[p] =$ "aborted", then $pending\,(s)\,[p] = pending\,(s_2)\,[p]$, else $pending\,(s)\,[p] = pending\,(s_1)\,[p]$;
5. the set $abortVals(s)$ is equal to $abortVals\,(s_2)$.

Note that if $\langle s_1, s_2\rangle$ and $s$ are related by the refinement mapping $f$, then $PendingReqs\,(s) = PendingReqs'$.

Under the refinement mapping $f$, the I/O automaton $SLin\,[1,i] \times SLin\,[i,i+1]$ simulates the I/O automaton $SLin\,[1,i+1]$ as follows.

1. Invoke and response actions of both $SLin\,[1,i]$ and $SLin\,[i,i+1]$ simulate, respectively, invoke and response actions of $SLin\,[1,i+1]$.
2. The $Recovery^i$ action of $SLin\,[i,i+1]$ simulates a $Linearize^1$ action of $SLin\,[1,i+1]$.
3. The switch actions $Switch^i$, which are the abort actions of $SLin\,[1,i]$ and the init actions of $SLin\,[i,i+1]$, are stuttering steps for $SLin\,[1,i+1]$.
4. The abort actions of $SLin\,[i,i+1]$, $Switch^{i+1}$, simulate abort actions of $SLin\,[1,i+1]$.
5. Both the $Linearize^1$ and the $Linearize^i$ actions simulate a $Linearize^1$ action of $SLin\,[1,i+1]$.

The most interesting cases are those of the $Recover^i$ action, the $Switch^j$ abort action of $SLin\,[i,i+1]$, and the $Linearize^i$ action of $SLin\,[i,i+1]$. The case of $Recover^i$ follows from invariant 3, the case of $Switch^j$ follows from invariants 4 and 5, and the case of $Linearize^i$ follows from invariant 6. □

For a more detailed proof please see the Isabelle/HOL formalization, which appears in section 5.5.

### 5.4.5 $SLin$ is a modular property

We have proved in the preceding sections that $SLin$ is linearizable and that $SLin$ is idempotent. To prove that $SLin$ is a modular property, it remains to show that $SLin$ is well-formed.

**Theorem 5.4** ($SLin$ is Well-Formed). *For every $j \in \mathbb{N}$, $SLin\,[j,j+1] \le ModeInst\,(j)$ and the I/O automata $SLin\,[1,j]$ and $SLin\,[j,j+1]$ are compatible.*

*Proof.* Consider the function $f$ which maps a state $s$ of $SLin\left[j,j+1\right]$ to the state $t$ of $ModeInst\left(j\right)$ by projecting $s$ onto its $status$ component, $f\left[s\right] = pending\left(s\right)$. The function $f$ is a refinement mapping from $SLin\left[i,i+1\right]$ to $ModeInst\left(i\right)$. Also note that the external signature of $SLin\left[i,i+1\right]$ is the same as the external signature of $ModeInst\left(i\right)$. Therefore, $SLin\left[i,i+1\right] \leq ModeInst\left(i\right)$.

It is easy to see that the I/O automata $SLin\left[1,i\right]$ and $SLin\left[i,i+1\right]$ are compatible by looking at their signatures. □

Finally, we can prove our main theorem.

**Theorem 5.5.** *The array of I/O automaton* $SLin\left[i,j\right]$, $i,j \in \mathbb{N}$, *is a modular property.*

*Proof.* Theorem 5.4 shows that $SLin\left[i,i+1\right]$ is a well-formed $i^{th}$ mode instance, corollary 5.1 shows that $SLin\left[i,i+1\right]$ is linearizable, and theorem 5.3 shows that $SLin\left[i,i+1\right]$ is idempotent. Therefore $SLin$ is a modular property. □

## 5.5 Proving Idempotence Mechanically

In an effort to make the results of this thesis trustworthy, we have mechanically proved the idempotence theorem in Isabelle/HOL. We briefly discuss our proof in this chapter and the full Isabelle/HOL theories ppear in appendix B.

Isabelle/HOL [76] is a highly trustworthy interactive proof assistant for higher order logic offering a sophisticated infrastructure. It is an instance of the generic interactive proof assistant Isabelle [80]. Isabelle/HOL allows writing and interactively proving statements in higher order logic. All proofs are checked by a small, highly trusted kernel of inference rules. A large library of derived proof rules and theorems is available and several packages provide automated setup for higher level concepts such as records, recursive and co-recursive data-types [89], recursive functions, modular organisation of specifications with locales [45], etc. The Isar proof language [91] allows writing structured and readable proofs in a style which is close to a detailed manual proof. Several automatic proof methods are available, such has the simplifier, the tableau prover [79], and Sledgehammer [8], which can call external automatic provers and SMT solvers [8] and reconstruct the obtained proofs in Isabelle/HOL. Moreover, the Nitpick tool [9] can search for counterexamples to putative theorems.

My Isabelle/HOL theories consist of a formalization of I/O-automata theory using finite traces, of a formalization of Recoverable Data-Type Representations, of a formalization of the SLin family of I/O automata, and of a full proof of the idempotence theorem.

The Isabelle/HOL proof of the idempotence theorem shows that $SLin\left[1,i\right] \times SLin\left[i,j\right]$ implements $SLin\left[1,j\right]$, for $1 < i < j$. The refinement mapping is essentially the same as in the proof of theorem 5.3. We prove the refinement mapping correct with the help of 25 state invariants about the composite automaton. The proof is written in the structured proof language Isar and consists of roughly 550 proof steps (i.e., lines containing the keyword "by"). With the specification, it forms a total of roughly 3900 lines of Isabelle/HOL code.

Our automata specification can be used as the basis for mechanically-checked refinement proofs of distributed protocols. Our proof of the idempotence theorem is a good example of such a refinement proof and shows that mechanically-checked proof of speculatively linearizable algorithms are feasible.

We conclude the chapter by a few subjective remarks on the author's experience with Isabelle/HOL. It is extremely time consuming for a relatively novice user to formalize and prove in Isabelle/HOL a theory that is not well-understood beforehand. The problem is that Nitpick and the other debugging tools available in Isabelle are not able to check high level properties like the idempotence or linearizability of $SLin$. Only deeply nested proof steps can be debugged in Isabelle/HOL. As a result, many errors where discovered late in the development and ultimately, although $SLin$ was proved idempotent after a lot of effort, the $SLin$ specification was found inadequate for proving $Quorum$. After this experience, the author formalized all his results in TLA+ and the TLC model checker was able to check our claims, end to end, before attempting any proof. Many errors were eliminated in the process, which culminated in a few month to the theory presented in this thesis. In contrast, our first development took more than a year and resulted in a mechanically checked proof of a property which is not exactly the right one in practice. In conclusion, even though experienced users may be able to use Isabelle/HOL effectively, the learning curve is still very steep for an outsider. However, debugging tools that allow quick prototyping are extremely useful and if integrated with Isabelle/HOL could allow a much broader audience to use it.

## 5.6 Conclusion

In this chapter we have presented the modular property $SLin$. Together with our model of adaptive algorithm the $SLin$ modular property forms the Speculative Linearizability framework.

We have introduced recoverable data-type representations (RDRs) and we have seen that the speculative linearizability property models systems in which the processes behave speculatively, i.e., they optimistically update a distributed implementation of the state of a RDR in a way that leads to increased performance under some optimistic assumptions and to the corruption of the state otherwise. If the state of the system is corrupted by an overly optimistic update, then the clients must detect it, abort their execution, and switch to the next mode, bringing along their estimate of a correct RDR state. Thanks to the properties of RDRs, the next modes can use the set of different RDRs received from the processes to recover a consistent RDR state and continue the execution in a linearizable fashion. Finally, note that, although we have given the responsibility to switch mode to clients, switching could in fact be performed by some agents different from the clients, such as the replica servers of a service provider.

In the next chapter we will see that the speculative linearizability property is efficiently implementable in the message-passing model of computation. To do so, we will present speculatively linearizable adaptive algorithms that efficiently implement any data type. We will also see in chapter 7 that speculative linearizability can be applied to the shared-memory model.

# 6 Applying Speculative Linearizability to Fault-Tolerant Message-Passing Systems

## 6.1 Introduction

In this chapter we apply speculative linearizability to build robust, linearizable, fault-tolerant message-passing algorithms. Thanks to speculative linearizability, we will obtain a new algorithm which improves upon the state of the art on several dimensions. We suppose that the clients in Π and a set of servers communicate through a fully-connected network. The relative speed of all the agents, clients and servers, and of the network are unknown and processes and servers can crash by stopping. An agent that does not crash executes its assigned algorithm faithfully. Our goal is to build a robust implementation of the data type $D$ in this environment, using the servers as internal components of the implementation.

Traditionally, fault-tolerant implementations of a data type were built using the State-Machine Replication technique, abbreviated SMR. In SMR, the servers, called replicas, each maintain a copy of the data-type representation. The servers use a sequence of independent instances of a consensus algorithm, where the first instance determines the first request to execute, the second instance determines the second request, and so on. Therefore, all the server execute the same sequence of requests and go through the same sequence of states. Thus, if a server crashes, then the clients can just use another one.

SMR works but has a drawback: because the requests are ordered by independent consensus instances, a SMR algorithm cannot easily optimize the execution of requests that commute. For example, even if the requests $r_1$ and $r_2$ commute, an SMR algorithm will guarantee that all servers agree on the same order between $r_1$ and $r_2$. However this is not necessary, because, by virtue of $r_1$ and $r_2$ commuting, any order results in the same outputs and future executions from the point of view of the clients.

The notion of Generalized Consensus [49] allows one to solve this problem. Generalized Consensus formalizes the task of agreeing, modulo the order of commuting requests, on a growing sequence of requests. Therefore Generalized Consensus is a specification of the problem that SMR is trying to solve, except that it has relaxed requirements for commuting requests. In contrast to SMR, Generalized Consensus does not mandate a specific implementation technique. In fact, SMR can be viewed as a possible implementation of Generalized Consensus, albeit one that does not take advantage of commutativity. In Generalized Consensus, the parties agreeing on the sequence of requests are the client, and not the servers. The servers are now part of the implementation of Generalized Consensus and need not follow any specific protocol a priori. Therefore, in contrast to SMR, there is no artificial separation between consecutive requests. Generalized Consensus is similar to linearizability but abstracts over how the processes should compute outputs, instead focusing on how to learn about the current state of the data type.

Generalized Paxos is an adaptive algorithm in the spirit of Fast Paxos [48] which implements Generalized Consensus. The servers of Generalized Paxos, called *acceptors*, execute a sequence of *ballots*, where each ballot can be either a fast ballot or a classic ballot. We will now call the servers “acceptors”. Both kinds of ballot may fail to make progress, leaving the task to a later ballot. Let us say that two requests are *non-conflicting* when either the two requests commute or the two requests are not invoked concurrently. The properties of Generalized Consensus allow a fast ballot to process non-conflicting requests with a latency of twice the communication delay between agents without relying on a leader process. In contrast, a classic ballot has a latency of three times communication delay between agents and relies on a correct leader process. However, under harsh conditions, classic ballots are more likely to make progress than fast ballots. The two types of ballots of Generalized Paxos can be seen as two modes of an adaptive algorithm.

Generalized Paxos only has two types of ballots. Moreover, ballots do not have a clear interface like mode instances and adding new ballot types is not easy. Multicoordinated Paxos [12] is an optimization Generalized Paxos which adds a new ballot type. The specification of Multicoordinated Paxos in TLA+ is more than 10 pages long [12]. Moreover, Multicoordinated Paxos is the only instance of optimization of Generalized Paxos that we know of, perhaps owing to the fact that, although Paxos is already notoriously hard to understand, Generalized Paxos is even more intricate than Paxos. Generalized Paxos is therefore not a robust algorithm.

In this chapter we present $QZ$, a new *robust* adaptive algorithm solving Generalized Consensus. The $QZ$ algorithm is obtained by combining two speculatively linearizable modes, namely $Quorum$ and $ZLight$, and has the following properties.

1. $QZ$ is robust: is it adaptive and, being speculatively linearizable, it can be composed with any other speculatively linearizable mode without any changes.
2. Progress is guaranteed when a strict majority of the acceptors are correct for a long enough time, like in Generalized Paxos.
3. $QZ$ can process non-conflicting requests with a delay of one message round-trip (including concurrent commuting requests), like Generalized Paxos.

Compared to Generalized Paxos, the main advantage of $QZ$ is that it can be easily extended with new modes.

In fact, to prove $QZ$ correct, we propose two intermediate specifications, $Fast\,(i)$ and $Safe\,(i)$, of what we call *fast* modes and *safe* modes. $Quorum$ refines the fast mode specification whereas $ZLight$ refines the safe mode specification .

Both the $Fast\,(i)$ and the $Safe\,(i)$ I/O automata can be seen as instances of Refined Quorum Systems [34]. The $Safe\,(i)$ I/O automaton uses quorums consisting of a strict majority of acceptors and the acceptors must not become inconsistent. A possible implementation of $Safe\,(i)$ would use a leader to ensure consistency, like $ZLight$. In contrast, the $Fast\,(i)$ I/O automaton uses bigger quorums to respond to requests but does not require consistency of the acceptors. In our abstract specifications of safe and fast modes, acceptors nondeterministically execute new requests, abstracting over the strategy used to coordinate the acceptors. Therefore one could use our abstract specifications to prove new safe or fast modes correct, such as a multi-coordinated safe mode in the spirit of Multicoordinated Paxos [12].

Another advantage of $QZ$ over Generalized Paxos is that $QZ$ can change the relative size of its types of quorum when changing mode instance. In Generalized Paxos, the relative size of the two types of quorums is fixed and changing it from one ballot to the next would break the algorithm. Changing the relative size of the different types of quorums is possible in $QZ$ by relying on at least one client to be correct, an assumption that Generalized Paxos does not make. However, in practice, if no client is correct then there is no point in running the system. Therefore we think that it is justified to assume that at least one client is correct. If clients cannot be trusted, then the service provider can setup special servers that play the role of clients just to change mode instance. In this case at least one of the servers playing the role of client should be correct in order for the system to make progress.

The $Quorum$ and $ZLight$ modes are generalizations, in the crash-stop fault model, of the algorithms of the same names proposed by Guerraoui et al.[32]. $Quorum$ is optimized for the execution of non-conflicting requests and can withstand less than one third of the acceptors crashing. It is fast even when requests are concurrent, as long as they commute. $ZLight$ works under contention even when requests conflict and can withstand less than half of the acceptors crashing. However it relies on a correct leader to make progress and will abort otherwise.

In the rest of this chapter we consider a dependency relation # of the data type $D$. We say that two requests $r_1$ and $r_2$ commute when $\langle r_1, r_2\rangle \notin \#$. As we have seen in section 3.2, the notion of “sequence of requests up to the order of commuting requests” is captured by the data-type representation $H^{\#}(D)$. In the rest of the chapter, we will therefore consider the data-type representation $\Delta = H^{\#}(D)$.

We work in the message-passing model with a fully connected network in which messages can be lost but not duplicated or corrupted in any way. On top of the client processes, we consider a set $A$ of $N$ *acceptor* processes.

## 6.2 Related Work

There are many fault-tolerant algorithms implementing Consensus, many of which could be considered variants of Paxos optimizing their performance according to different metrics or under different assumptions. The following algorithms are examples in the crash-stop fault model: Ring Paxos [65], Multi-Ring Paxos [64], Fast Paxos [48], Disk Paxos [29], Egalitarian Paxos [73], Multi-Coordinated Paxos [12], Vertical Paxos [55], Cheap Paxos [56], Paxos-MIC [41], Mencius [63], and Fast Mencius [90]. In the Byzantine model, examples of algorithms based on Paxos include FaB Paxos [66], Zyzzyva, [46], PBFT [14], Aardvark [19], Q/U [1], and HQ [22].

The Abstract framework [35] allows building adaptive Byzantine fault-tolerant algorithms out of independent modules. The Aliph algorithm is an adaptive Byzantine fault-tolerant algorithm built in the Abstract framework. Aliph uses three types of modules among which are Byzantine versions of the $Quorum$ and $ZLight$ algorithms presented in this chapter. The speculative linearizability framework, presented in chapter 5, is inspired from the Abstract framework.

Generalized Paxos [49] is an adaptive fault-tolerant algorithm that optimizes the execution of commuting requests and that can switch between two different modes of execution. The algorithm uses the concept of ballot, which can be either a fast ballot, in which case an optimistic mode is used, or a classic ballot, in which a mode similar to the original Paxos is used. The consistency across ballots is ensured by some invariants, notably about the size of the intersection of the quorums that the two modes use. In principle, other types of modes could be used if they preserve these invariants. However, in contrast to our work, there is no clearly identified interface for adding new ballot types to the algorithm.

Most if not all of the algorithms cited above rely on the notion of a quorum. A quorum is a set of servers big enough to reliably hold information despite of the failures allowed by the computing model. The trade-offs between the tolerated number of faults, the nature of faults (Byzantine faults or crashes), and the latency to respond to a request are captured in Refined Quorum System [34]. Lower bounds relating the latency and size of quorums in the crash-stop model are rigorously proved in [50].

## 6.3 Fast and Safe Modes

In this section we present the specifications of fast and safe modes, which are both speculatively linearizable. Those specifications abstract over the communication between processes and over the strategy used to coordinate acceptors: the state of every process is readable by every other process and acceptors nondeterministically execute new requests. One can refine fast or safe modes by implementing the state accesses and coordination using the network, obtaining a concrete algorithm. For example, $Quorum$ refines the fast mode I/O automaton and $ZLight$ refines the safe mode I/O automaton.

TLA+ versions of the specification of fast and safe mode appear in appendix A, in modules FastMPGC and SafeMPGC. Both modules use the MPGC module (MPGC stands for Message-Passing Generalized Consensus), which factorizes the common structure of fast and safe modes.

The two I/O automata $Safe(i)$ and $Fast(i)$ have the external signature of a speculatively linearizable $i^{th}$ mode instance. The input actions of $Safe\ (i)$ and $Fast\ (i)$ are the actions of the form $Switch_p^i\ (c, iv)$ or $Inv_p^i\ (c)$ where $p$ is a client, $c$ is a command, and $iv$ is a switch value (a Δ-state) except when $i = 1$, in which case there are not input switch actions. The output actions of $Safe\ (i)$ and $Fast\ (i)$ are the actions the form $Switch_p^{i+1}\ (c, av)$ or $Resp_p^i\ (o)$ where $p$ is a client, $c$ is a command, $o$ is an output, and $av$ is a switch value (a Δ-state).

Both $Safe\ (i)$ and $Fast\ (i)$ have the same set of states:

1. a set $initVals$, tracking the init values received;
2. a request $pending\ [p]$, tracking the pending request of $p$, for every client $p$;
3. the status of the client $p$, $status\ [o]$, for every client $p$;
4. the status of the acceptor $a$, $accStatus\ [a]$, for every acceptor $a$;
5. the local Δ-state of the acceptor $a$, $dState\ [a]$, which can be a Δ-state or the special value $None$, for every acceptor $a$.

On top of the actions of their external signature, i.e., the invocations, responses, init actions, and abort actions, the two I/O automata have five types of internal actions: $Panic\ (p)$, where $p$ is a client, and, for every acceptor $a$, $Exec\ (a)$, $WakeUp\ (a)$, $TimeOut\ (a)$, and $Stop\ (a)$.

Clients are in status “idle”, “ready”, “pending”, “panic”, or “aborted”. A $Panic\ (c)$ action brings a client $p$ from the status “pending” to the status “panic”, at which point $p$ may later abort. As in $SLin\ (i, i + 1)$, if $i = 1$, then every client is initially ready; otherwise, every client is initially sleeping.

The acceptors also have a status, which is either “idle”, “ready”, or “stopped”. If $i = 1$, then every acceptor is initially “ready”; otherwise, every acceptor is initially “idle”. After a $WakeUp\ (a)$ action, the acceptor becomes ready. After a $Stop\ (a)$ action, the acceptor $a$ is stopped. Finally, for every acceptor $a$, $dState\ [a]$ is initially ⊥.

### 6.3.1 Behavior of The $Safe\ (i)$ I/O automaton

To make progress, the $Safe\ (i)$ I/O automaton relies on a *safe quorum* of acceptors to be correct. The safe quorums are the sets of acceptors such that the intersection between any two safe quorums is nonempty. This translates to the following definition of safe quorums.

$$SafeQuorum = \left\{ Q \subseteq A : Card\left(Q\right) \geq \left\lfloor \frac{N}{2} \right\rfloor + 1 \right\} \tag{6.1}$$

The acceptors are said *consistent* when for every two acceptors $a_1$ and $a_2$, either $dState\ [a_1]$

is a prefix of $dState\,[a_2]$ or $dState\,[a_2]$ is a prefix of $dState\,[a_1]$. The $Safe\,(i)$ I/O automaton ensures that the acceptors are always consistent. However it abstracts over the implementation of this guarantee, leaving as much freedom as possible to the implementations. In practice, the guarantee can be implemented with a leader, as in $ZLight$, but other implementations are possible.

We now describe the actions of the $Safe\,(i)$ I/O automaton.

1. An init action $Switch_p^i\,(c,v)$ is enabled when the client $p$ is in status “idle”, which is possible only if $i > 1$. The action adds $v$ to the set $initVals$, sets $pending\,[p]$ to $\langle p,c\rangle$, and sets the status of $p$ to “pending”.
2. An invocation action $Inv_p^i\,(c)$ is enabled when the client $p$ is ready. The action sets $pending\,[p]$ to $\langle p,c\rangle$ and sets the status of $p$ to “pending”.
3. $WakeUp\,(a)$, executed by an acceptor $a$, is enabled if $a$ is “idle” and if there exists $iv \in initVals$ such that substituting $iv$ for the value of $dState\,[a]$ would leave the acceptors in a consistent state. The effect of the action is to perform the substitution and to set the status of $a$ to “ready”.
4. $Exec\,(a)$ is enabled when the acceptor $a$ is “ready”, a client $p$ has a pending request $\langle p,c\rangle$, and if substituting $dState\,[a] \bullet c$ for the value of $dState\,[a]$ would leave the acceptors in a consistent state. The effect of the action is to perform the substitution. In leader-based algorithms, the action models an acceptor receiving the next request to execute from the leader.
5. A response action $Resp_p^i\,(o)$ is enabled when the client $p$ has a pending request $r$ and there is a safe quorum $Q$ of acceptors which are not idle and whose set of Δ-states $S_Q$ is such that $GLB\,(S_Q)$ contains $r$ and $o = \gamma(GLB\,(S_Q), r)$. The effect of the action is to set the status of $p$ to “ready”.
6. $Panic\,(p)$, executed by a client $p$, is enabled when $p$ is in status “pending”. The effect of the action is to set the status of $p$ to “panic”. In leader-based algorithms, the action models the client $p$ detecting a faulty leader and initiating a mode change.
7. $Stop\,(a)$, executed by an acceptor $a$, is enabled when there is a client $p$ which has panicked. The effect of the action is to set the status of the acceptor $a$ to “stopped”, preventing it from executing any new requests. The action models the acceptor $a$ receiving through the network a notification that the client $p$ has panicked.
8. $TimeOut\,(a)$, executed by an acceptor $a$, is enabled when $a$ is in status “idle”. The effect of the action is to set the the local Δ-state of $a$ to the special value $None$ and to set the status of $a$ to “stopped”. In leader-based algorithms, this action allows the acceptors to stop despite never having heard from a faulty leader.
9. The abort action $Switch_p^{i+1}\,(c, av)$ is enabled when $p$ has panicked, $pending\,[p] = \langle p,c\rangle$, and
   - there exists a safe quorum $Q$ of acceptors whose Δ-state is not $None$, which have all stopped, and such that the maximal Δ-state of the acceptors in $Q$ is the abort values $av$, or

– there exists a safe quorum $Q$ of acceptors whose Δ-state is equal to $None$ and which have all stopped, and $av$ is a member of $initVals$.

The effect of the action is to set the status of $p$ to "aborted". The action models $p$ aborting when it has received from every acceptor in $a \in Q$ an acknowledgement that $a$ has stopped along with the Δ-state of $a$ and using an abort value equal to the maximal Δ-state received. Note that because a safe mode guarantees that the acceptors are consistent, a quorum of acceptors always has a maximum Δ-state.

The $Safe\,(i)$ I/O automaton simulates the $SLin\,[i, i+1]$ I/O automaton in a simple way. First add a history variable $abortVals$ which is initialized to the empty set and which is updated on every abort action by adding the abort value to the set. Then $Safe\,(i)$ refines $SLin\,[i, i+1]$ under the refinement mapping $f$ associating a state of $s$ of $Safe\,(i)$ to the state of $t$ of $SLin\,[i, i+1]$ as follows.

1. Every client $p$ has the same status in $t$ as in $s$ except that when $p$ has panicked in $s$, in which case $p$ is in status "pending" in $SLin\,[i, i+1]$.
2. Every client has the same pending request in $s$ and $t$.
3. If $1 < i$, then the boolean $initialized(t)$ be true if and only if there is, in $s$, a safe quorum of acceptors which are not idle. If $i = 1$ then $initialized(t)$ is always true.
4. The Δ-state $dState(t)$ is the maximum over all non-idle safe quorums $Q$, in which no acceptor has a local Δ-state equal to $None$, of the glb of the Δ-states of the members of $Q$:

$$\begin{aligned} s = Max\, \big\{ &GLB\left(\{dState\,[a] : a \in Q\}\right) : \\ &Q \in SafeQuorum \wedge \forall a \in Q : status\,[a] \neq idle \wedge dstate\,[a] \neq None \big\}. \end{aligned} \tag{6.2}$$

5. The sets $initVals(t)$ and $abortVals(t)$ are equal, respectively, to $initVals(s)$ and $abortVals\,(s)$.

The most interesting case of the proof of refinement, had we formalized it, would be the abort action. In this case we need to show that the abort value is a safe abort value. We show that the abort value is an extension with pending requests of the global Δ-state $dState(t)$ of $SLin\,[i, i+1]$. By definition of $f$, we know that the global Δ-state $dState(t)$ is the glb of the Δ-states of a safe quorum $Q$ of acceptors. Therefore, every acceptor $a$ of $Q$ has a Δ-state $dState(s)\,[a]$ greater than or equal to $dState\,(t)$. By property of safe quorums, any other safe quorum $R$ has a member $b \in R \cap Q$. Moreover, because the acceptors are always consistent, every acceptor $c \in R$ is such that $dState\,(s)\,[c]$ is a prefix of $dState\,(s)\,[b]$ or vice versa. Therefore the maximum $m$ over the acceptors $c \in R$ of $dState\,[c]$ is an extension of $dState(t)$. Finally, the acceptors only execute pending requests, so the $m$ is an extension of $dState(t)$ with pending requests.

The TLA+ formalization of the $Safe\,(i)$ I/O automaton and of the refinement mapping can be found in appendix A. The refinement has been model checked exhaustively with TLC using the consensus data type with four acceptors, three clients, and two consensus values, and with the generic data type with three acceptors, two clients, a unique command, and sequences of length smaller than or equal to 3.

### 6.3.2 Behavior of The $\mathit{Fast}\,(i)$ I/O automaton

To compute the output to its request, a client of the $\mathit{Fast}\,(i)$ I/O automaton communicates with a *fast quorum* of acceptors. In contrast to the safe quorums of $\mathit{Safe}\,(i)$, the Δ-states of a fast quorum of acceptors can become inconsistent, allowing implementations in which clients communicate directly with each acceptor, without the intermediary of a leader, and get a response to their request with a latency of two communication delays. But, to allow safe aborts when the Δ-states of the acceptors become inconsistent, fast quorums have to be bigger than safe quorums. Still, only a smaller type of quorum, *recovery quorums*, is needed in order for fast implementations to eventually abort. To sum up, in a fast mode, a client needs to communicate with a fast quorum of acceptors in order to determine a response to its request and a client needs to communicate with a recovery quorum of acceptors in order to determine an abort value and switch mode. Fast quorums and recovery quorums must satisfy the following constraints:

1. If $Q$ and $R$ are two fast quorums, then $Q \cap R \neq \emptyset$.
2. If $Q$ is a fast quorum and $R$ is a recovery quorum, then the intersection of $Q$ and $R$ consists of a strict majority of the members of $R$:

$$\mathit{Card}\left(Q \cap R\right) \geq \left\lfloor \frac{\mathit{Card}\,(R)}{2} \right\rfloor + 1. \tag{6.3}$$

Fast quorums and recovery quorums have been described before in the context of Refined Quorum Systems [34]. Lower bounds on the size of quorums for solving asynchronous consensus are given in [50]. To satisfy the constraints on the intersection of quorums, we can take the following definitions of fast and recovery quorums:

$$\mathit{FastQuorum} = \left\{Q \subseteq A : \mathit{Card}\left(Q\right) \geq \left\lfloor \frac{2N}{3} \right\rfloor + 1\right\} \tag{6.4}$$

$$\mathit{RecoveryQuorum} = \left\{Q \subseteq A : \mathit{Card}\left(Q\right) \geq \left\lfloor \frac{2N}{3} \right\rfloor + 1\right\} \tag{6.5}$$

or

$$\mathit{FastQuorum} = \left\{Q \subseteq A : \mathit{Card}\left(Q\right) \geq \left\lfloor \frac{3N}{4} \right\rfloor + 1\right\} \tag{6.6}$$

$$\mathit{RecoveryQuorum} = \left\{Q \subseteq A : \mathit{Card}\left(Q\right) \geq \left\lfloor \frac{N}{2} \right\rfloor + 1\right\} \tag{6.7}$$

or

$$\mathit{FastQuorum} = \{A\} \tag{6.8}$$

$$\mathit{RecoveryQuorum} = \{\{a\} : a \in A\} \tag{6.9}$$

The transitions of the $\mathit{Fast}\,(i)$ I/O automaton are similar to the ones of the $\mathit{Safe}\,(i)$ I/O automaton. The $\mathit{Exec}\,(a)$, $\mathit{Resp}\,(a)$, and $\mathit{WakeUp}\,(a)$ actions are identical to the ones of the $\mathit{Safe}\,(i)$ I/O automaton, except that the consistency condition is removed and fast quorums are substituted for safe quorums. Therefore, the Δ-states of the acceptors can become inconsistent, meaning there may be two Δ-states such that neither is the prefix of the other.

### Aborting in $Fast\,(i)$

The abort action has to be changed more significantly in order to allow aborting when the acceptors are inconsistent. A client running the $Safe\,(i)$ I/O automaton aborts with the maximum Δ-state of a safe quorum of acceptors. In the $Fast\,(i)$ I/O automaton, the set of Δ-states of a recovery quorum of acceptors may not have a maximum if the acceptors are inconsistent.

The $Fast\,(i)$ I/O automaton refines the $SLin\,[i, i+1]$ I/O automaton under the same refinement mapping $f$ as the $Safe\,(i)$ I/O automaton except that fast quorums are substituted for safe quorums. Let us see how to modify the abort action in order for the refinement mapping to hold.

Consider an abort step $\left\langle s, Switch_p^{i+1}\,(c, av)\,, s'\right\rangle$ of $Fast\,(i)$ and the states $t = f\,[s]$ and $t' = f\,[s']$. For $\left\langle t, Switch_p^{i+1}\,(c, av)\,, t'\right\rangle$ to be an abort step of $SLin\,[i, i+1]$, we must show that $av$ is a safe abort value $av \in SafeAborts\,(t)$.

By definition of the refinement mapping $f$, there is a fast quorum $Q$ such that $dState\,(t)$ is the glb of the Δ-states of the acceptors in $Q$. Therefore, every member of $a \in Q$ has a larger Δ-state than $dState\,(t)$:

$$\forall a \in Q : dState\,(s)\,[a] \geq dState\,(t) \tag{6.10}$$

By property of recovery quorums, for every recovery quorum $R$, the set of acceptors $Q \cap R$ consists of a strict majority of $R$. Therefore, in every strict majority $M$ of the members of $R$, there is one acceptor $b \in M \cap Q$. By eq. (6.10), $dState\,(s)\,[b]$ is an extension of $dState\,(t)$. Therefore, either the glb $g_M$ of the Δ-states of the acceptors in $M$ is a prefix of $dState\,(t)$, or it is an extension of $dState\,(t)$. Moreover, if we take $M = R \cap Q$, then $g_M$ is an extension of $dState\,(t)$.

To sum up, for every recovery quorum $R$,

1. for every strict majority $M$ of $R$, the glb of the Δ-states of $M$ is either a prefix or an extension of $dState\,(t)$;
2. the set of acceptors $Q \cap R$ is a strict majority of $R$ and the glb of $Q \cap R$ is an extension of $dState\,(t)$.

Let $MajSets\,(R)$ be the set of majorities of $R$. Define $G\,(R)$ as the set obtained by taking, for every $M \in MajSets\,(R)$, the glb of the Δ-states of the acceptors in $M$,

$$G\,(R) = \left\{GLB\left(\left\{dState\,[a] : a \in as\right\}\right) : as \in MajSets\,(R)\right\} \tag{6.11}$$

Finally, define the abort values determined from $R$ as the members of $G$ which have no extension in $G$,

$$AbortValues\,(R) = \{g \in G\,(R) : \forall g' \in G\,(R) \setminus \{g\} : \neg g \preceq g'\} \tag{6.12}$$

From the properties listed above in items 1 and 2, we can conclude that any member of $AbortValues\,(R)$ is of the form $dState\,(t) \star rs$, where $rs$ is a sequence of pending requests. Therefore, to abort, a client choses a recovery quorum $R$ and uses an arbitrary value in $AbortValues\,(R)$ as abort value.

We now describe the complete transition relation of the $Fast\,(i)$ I/O automaton.

1. An init action $Switch_p^i\,(c, v)$ is enabled when the client $p$ is in status "idle", which is possible only if $i > 1$. The action adds $v$ to the set $initVals$, sets $pending\,[p]$ to $\langle p, c\rangle$, and sets the status of $p$ to "pending".
2. An invocation action $Inv_p^i\,(c)$ is enabled when the client $p$ is in status "ready". The action sets $pending\,[p]$ to $\langle p, c\rangle$ and sets the status of $p$ to "pending".
3. $WakeUp\,(a)$, executed by an acceptor $a$, is enabled if $a$ is "idle" and $initVals$ in nonempty. The effect of the action is to set $dState\,[a]$ to one of the $\Delta$-states in $initVals$ and to set the status of $a$ to "ready". Note that there is no constraint on the init value chosen to update $dState\,[a]$.
4. $Exec\,(a)$ is enabled when the acceptor $a$ is "ready" and there is a client $p$ which has a pending request $\langle p, c\rangle$. The effect of the action is to execute the request $\langle p, c\rangle$ locally by updating $dState\,[a]$ to $dState\,[a] \bullet \langle p, c\rangle$.
5. A response action $Resp_p^i\,(o)$ is enabled when the client $p$ has a pending request $r$ and there is a fast quorum $Q$ of acceptors which are not idle and whose set of $\Delta$-states $S_Q$ is such that $GLB\,(S_Q)$ contains $r$ and $o = \gamma(GLB\,(S_Q), r)$. The effect of the action is to set the status of $p$ to "ready".
6. $Panic\,(p)$, executed by a client $p$, is enabled when $p$ has a pending request. The effect of the action is to set the status of $p$ to "panic". The action models the client $p$ detecting an inconsistent fast quorum of acceptors and initiating a mode change.
7. $Stop\,(a)$, executed by an acceptor $a$, is enabled when there is a client $p$ which has panicked. The effect of the action is to set the status of the acceptor $a$ to "stopped", preventing it from executing any new requests. The action models the acceptor $a$ receiving through the network a notification that the client $p$ has panicked.
8. The abort action $Switch_p^{i+1}\,(c, av)$ is enabled when $p$ has panicked, $pending\,[p] = \langle p, c\rangle$, and there exists a recovery quorum $R$ of acceptors which have stopped and such that $av \in AbortValues\,(R)$. The effect of the action is to set the status of $p$ to "aborted". The action models the client $p$ aborting when it has received from every acceptor in $a \in R$ an acknowledgement that $a$ has stopped along with the $\Delta$-state of $a$.

Similarly to the $Safe\,(i)$ I/O automaton, the $Fast\,(i)$ I/O automaton simulates the $SLin\,[i, i+1]$ I/O automaton. The refinement mapping is the same, adding the same $abortVals$ history variable, except that fast quorums are substituted for safe quorums. The refinement mapping was model-checked exhaustively with TLC using the consensus data type with four acceptors, three clients, and two consensus values, and with the generic data type with three acceptors, two clients, a unique command, and sequences of length smaller than or equal to 3.

We have seen that both the safe mode specification and fast mode specification are speculatively linearizable. Therefore, any concrete mode refining either the safe mode specification or the fast mode specification is also speculatively linearizable and can be combined with any other speculatively linearizable mode.

The TLA+ specifications of $Safe\,(i)$ and $Fast\,(i)$ can be found in appendix A.

We will now present the $Quorum$ and $ZLight$ modes and show that $Quorum$ refines the fast mode and $ZLight$ refines the safe mode. We will also see that the $QZ$ adaptive algorithm, obtained by combining $Quorum$ and $ZLight$, has the same progress guarantee as Generalized Paxos and can execute non-conflicting requests with a latency of two communication delays.

## 6.4 The $QZ$ Algorithm

In this section we present the $Quorum$ and $ZLight$ modes and the adaptive algorithm

$$QZ = \{Quorum, ZLight\}. \tag{6.13}$$

The $Quorum\,(i)$ I/O automaton refines the fast mode I/O automaton $Fast\,(i)$, whereas $ZLight\,(i)$ refines the safe mode I/O automaton $Safe\,(i)$. The $QZ$ adaptive algorithm has the similar progress guarantees as Generalized Paxos: invocations are eventually given a response if there eventually is a recovery quorum of acceptors which is correct for a long enough time.

The TLA+ version of the specification of $Quorum$ and $ZLight$ appear in appendix A.

### 6.4.1 $Quorum$

The $Quorum$ algorithm implements the fast mode specification by concretely specifying how new requests and the Δ-states of the acceptors are propagated through the network.

A client that invokes a request simply broadcasts it to all of the acceptors. An acceptor that receives a new request executes it immediately, without synchronization with the other acceptors, and sends its new Δ-state to the client that issued the request. A client returns a response when it has received the Δ-states of a fast quorum of acceptors, provided the glb of the received Δ-states contains its request.

If the glb of the received Δ-states does not contain its requests, then the acceptors have become inconsistent because, for lack of synchronization, they have executed non-commuting requests in different orders. In this situation, $Quorum$ cannot make progress any more and clients must abort. To abort with a safe abort value, a client “panics” by broadcasting a “panic" message to all the acceptors. The client then waits for an acknowledgement, containing the local Δ-state of the sender, from a recovery quorum of acceptors. Once the needed acknowledgements have been received, the client computes an abort value as in the $Safe\,(i)$ I/O automaton and switches to the next mode instance. When an acceptor receives its first "panic" message, it stops executing new requests, mimicking the $Stop\,(a)$ action of the safe mode specification.

Let us now describe the I/O automaton $Quorum\,(i)$ in more details. For simplicity, the $Quorum\,(i)$ I/O automaton is a monolithic I/O automaton, i.e., it is not obtained by composing individual I/O automata corresponding to each agent in the system.

The signature of the $Quorum\,(i)$ I/O automaton is the same as the one of the $Fast\,(i)$ I/O automaton with the addition of two internal actions $RcvExecAck\,(p)$ and $RcvPanicAck\,(p)$, for every client $p$. Therefore, the input actions of $Quorum\,(i)$ are the actions the form $Switch_p^i\,(c, iv)$ or $Inv_p^i\,(c)$ where $p$ is a client, $c$ is a command, and $iv$ is a switch value (a Δ-state) except when $i = 1$, in which case there are not input switch actions. The output actions of $Quorum\,(i)$ are the actions the form $Switch_p^{i+1}\,(c, av)$ or $Resp_p^i\,(o)$ where $p$ is a client, $c$ is a command, $o$ is an output, and $av$ is a switch value (a Δ-state). The internal actions of $Quorum\,(i)$ are the actions of the form $Panic\,(p)$, $RcvExecAck\,(p)$, $RcvPanicAck\,(p)$ where $p$ is a client and $Exec\,(a)$, $WakeUp\,(a)$, and $Stop\,(a)$, where $a$ is an acceptor.

The states of the $Quorum\,(i)$ I/O automaton are the same as the $Fast\,(i)$ I/O automaton with the addition, for every client $p$, of two arrays $execAcks\,[p]\,[a]$ and $panicAcks\,[p]\,[a]$ mapping every acceptor $a$ to a Δ-state. The other components, the ones inherited from $Fast\,(i)$, are $status\,[p]$, $pending\,[p]$, $initVals$, $accStatus\,[a]$, and $dState\,[a]$. As in $Fast\,(i)$, for every client $p$, $status\,[p]$ is the control flow location of $p$; for every client $p$, $pending\,[p]$ contains the pending requests of $p$ if it has one; $initVals$ contains the set of init values that appeared so far; for every acceptor $a$, $dState\,[a]$ contains the local Δ-state of $a$ and $accStatus\,[a]$ is the control flow location of $a$.

The state of $Quorum\,(i)$ also has a network component that we will not explicitly describe. However, the network allows any client or acceptor to send or receive messages to and from other clients or acceptors.

Initially, for every client $c$, $execAkcs[c]$ and $panicAcks[c]$ map every acceptor to the special value $none$. Moreover, as in $Safe\,(i)$, if $i > 1$ then every client or acceptor is initially "idle". Otherwise, when $i = 1$, every client and acceptor is initially "ready" and every acceptor has a local Δ-state equal to $\perp$.

We now describe in detail the actions of the $Quorum\,(i)$ I/O automaton.

1. An init action $Switch_p^i\,(c, v)$ is enabled when the client $p$ is not initialized, which is possible only if $i > 1$. The action adds $v$ to the set $initVals$, sets $pending\,[p]$ to $\langle p, c\rangle$, and broadcasts the messages $\langle "init", v\rangle$ and $\left\langle "req", \langle p, c\rangle\right\rangle$ to all the acceptors.
2. An invocation action $Inv_p^i\,(c)$ is enabled when the client $p$ is ready. The action sets

$pending[p]$ to $\langle p, c\rangle$ and broadcasts the message $\langle \text{"req"}, \langle p, c\rangle\rangle$ to all the acceptors.

3. $WakeUp(a)$, executed by an acceptor $a$, is enabled if $a$ is "idle" and $a$ can receive an $\langle \text{"init"}, v\rangle$ message from a client. The effect of the action is to receive the message and to set $dState[a]$ to $v$.
4. $Exec(a)$ is enabled when $a$ is ready and $a$ can receive a $\langle \text{"req"}, \langle p, c\rangle\rangle$ message from a client. The effect of the action is to receive the message, to set $dState[a]$ to $dState[a] \bullet \langle p, c\rangle$, and to send the message $\langle \text{"execAck"}, dState[a] \bullet \langle p, c\rangle\rangle$ to $p$.
5. $RcvExecAck(p)$ is enabled when the client $p$ can receive a message $\langle \text{"execAck"}, v\rangle$ from an acceptor $a$. Its effect is to receive the message and to set $execAcks[p][a]$ to $v$.
6. A response action $Resp_p^i(o)$ is enabled when there exists a fast quorum $Q$ of acceptors such that, for every $a \in Q$, $p$ has received an acknowledgement from $a$, the glb
$$g = GLB(\{execAcks[p][a] : a \in Q\}) \tag{6.14}$$
of the acknowledgements contains the pending request of $p$, and $o = \gamma(g, pending[p])$.
7. $Panic(p)$, executed by a client $p$, is enabled when $p$ has a pending request. Its effect is to broadcast the message $\langle \text{"panic"}\rangle$ to all the acceptors. Note that a client can panic at any time, even if it did not detect any inconsistency in the acceptor's replies. Therefore, the switching policy of the system can instruct $Quorum$ to abort for performance reasons at any time.
8. $Stop(a)$, executed by an acceptor $a$, is enabled when $a$ can receive a $\langle \text{"panic"}\rangle$ message from a client $p$. Its effect is to receive the message, to stop $a$, which will not execute any more requests, and to send the message $\langle \text{"panicAck"}, dState[a]\rangle$ to $p$.
9. $RcvPanicAck(p)$ is enabled when $p$ has panicked and can receive a $\langle \text{"panicAck"}, v\rangle$ message from an acceptor $a$. Its effect is to receive the message and to set $panicAcks[p][a]$ to $v$.
10. The abort action $Switch_p^{i+1}(c, av)$ is enabled when $p$ has panicked, $pending[p] = \langle p, c\rangle$, and there exists a recovery quorum $R$ of acceptors such that $p$ has received a panic acknowledgement from every member of $R$, and $av$ is computed as explained in the description of the $Fast(i)$ I/O automaton.

$Quorum$ refines the $Fast(i)$ I/O automaton: the refinement mapping simply consists in projecting the state of $Quorum$ onto the state of $Fast(i)$, erasing the components that are not part of the state of $Fast(i)$. The refinement mapping has been checked by TLC using the consensus data type with four acceptors, three clients, and two consensus values, and with the generic data type with three acceptors, two clients, a unique command, and sequences of length smaller than or equal to three.

### Progress Guarantees

We now informally discuss the liveness properties of the $Quorum$ mode. The discussion can only be informal because our formal model, being restricted to finite traces, cannot model liveness properties.

We can see that, to respond to a request, a client needs to receive acknowledgements from a fast quorum of acceptors. However, a client can panic at any time and then abort when it has received acknowledgements from a recovery quorum of acceptors. Therefore, if, eventually, a recovery quorum of acceptors is correct for a long enough time, then a client can eventually abort. If a fast quorum is correct then a client will eventually get a response to its invocation.

Finally, note that if two requests commute, then, even if they are executed in different orders by different acceptors, $Quorum$ will not abort and will process them with a latency of two communication delays. This is because executing two commuting requests always results in the same state, independently of the order of their execution.

### 6.4.2 $ZLight$

Remember that safe modes use smaller quorums than fast modes but must ensure that the local Δ-states of the acceptors remain consistent. The $ZLight$ algorithm relies on a distinguished acceptor, called the leader, to enforce the consistency requirement of safe modes. The leader can be different in different instances of $ZLight$.

In contrast to $Quorum$, a client of $ZLight$ does not broadcast its request but sends it only to the leader. The leader then executes the request and broadcasts its new Δ-state to all the other acceptors. An acceptor updates its local Δ-state to any bigger Δ-state received from the leader. Both the leader and the other acceptors send their new Δ-states to the clients in acknowledgment messages. A client produces a response when it has received Δ-states from a safe quorum of acceptors. If the leader is faulty, the acceptors may never be able to initialize their local Δ-state and a client may never receive enough Δ-states from the acceptors. Therefore, at any point, a client can abort by triggering a “panic” process, similarly as in the $Quorum$ algorithm, and an acceptor can time out. An acceptor that times out irrevocable sets its local Δ-state to a special value, $None$, that is not a Δ-state. A client that aborts broadcasts a panic message to all the acceptors and waits for acknowledgments, containing local Δ-states, from a safe quorum of acceptors. Once all the needed acknowledgments have been received, the client computes an abort value as in the I/O automaton $Safe\,(i)$ and switches to the next mode.

Let us now describe the I/O automaton $ZLight\,(i)$ in more details. The signature and the states of the $ZLight\,(i)$ I/O automaton are the same as the ones of the $Quorum\,(i)$ I/O automaton. The actions of $ZLight\,(i)$ differ from the actions of $Quorum\,(i)$ in the way that clients send their requests to the acceptors, through the intermediary of a leader in $ZLight$, in the types of quorums used, and in the way that an aborting client computes its abort value. We suppose the existence of a distinguished acceptor $leader\,(i)$. The actions of the $ZLight\,(i)$ I/O automaton are obtained by modifying those of the $Quorum\,(i)$ I/O automaton as follows.

1. In an init action $Switch_p^i\left(p,v\right)$, the client $c$ sends its $\langle\text{"init"},v\rangle$ and $\left\langle\text{"req"},\langle p,c\rangle\right\rangle$ mes-

sages only to the leader *only*, instead of broadcasting to all the acceptors.

2. In an invocation action $Inv_p^i(p)$ the client $c$ sends its $\langle$"req", $\langle p, c\rangle\rangle$ message *only* to the leader, instead of broadcasting it to all the acceptors.
3. $WakeUp(leader(i))$ is as in $Quorum$ (the leader is also an acceptor) except that, on top of sending an acknowledgement to the client, the leader broadcasts the message $\langle$"leader-init", $v\rangle$ to all the other acceptors.
4. The $Exec(leader(i))$ action is as in $Quorum$ except that, on top of sending an acknowledgement to the client $p$ sending the request, the leader broadcasts the message $\langle$"leader-exec", $p, dState'[leader(i)]\rangle$ to the other acceptors, where $dState'[leader(i)]$ is the new $\Delta$-state of the leader.
5. $WakeUp(a)$, where $a \neq leader(i)$ is an acceptor, is enabled when $a$ is idle and can receive a message $\langle$"leader-init", $v\rangle$ from the leader. The effect of the action is to receive the message and to set $dState[a]$ to $v$.
6. $Exec(a)$, where $a \neq leader(i)$ is an acceptor, is enabled when $a$ is ready and $a$ can receive a $\langle$"leader-exec", $p, v\rangle$ message from the leader. The effect of the action is to receive the message and, only if $dState[a] \leq v$, to set $dState[a]$ to $v$ and to send the message $\langle$"execAck", $v\rangle$ to the client $p$.
7. $RcvExecAck(p)$ is exactly as in $Quorum$.
8. A response action $Resp_p^i(o)$ is as in $Quorum$ except that a safe quorum is substituted for the fast quorum.
9. $Panic(p)$, $RcvPanicAck(p)$, and $Stop(a)$ are exactly the same as in $Quorum$.
10. $TimeOut(a)$ is the same as in fast modes. The action is executed by an acceptor $a$, is enabled when $a$ is in status “idle”. The effect of the action is to set the local $\Delta$-state of $a$ to the special value $None$ and to set the status of $a$ to “stopped”.
11. The abort action $Switch_p^{i+1}(c, v)$ is enabled when the client $p$ has panicked, $pending[p] = \langle p, c\rangle$, there exists a safe quorum $R$ of acceptors such that $p$ has received a panic acknowledgement from every member of $R$, and
    - for every member $a$ of $R$, $panicAcks[p][a]$ is not $None$ and the maximal $\Delta$-state of $\{panicAcks[p][a] : a \in R\}$ is the abort values $av$, or
    - for every member $a$ of $R$, $panicAcks[p][a] = None$ and $av$ is a member of $initVals$.

$ZLight(i)$ refines the $Safe(i)$ I/O automaton: the refinement mapping simply consists in projecting the state of $ZLight$ onto the state of $Safe(i)$, erasing the components that are not part of the state of $Safe(i)$. $ZLight(i)$ respects the consistency property of $Safe(i)$ because acceptors only update their state when instructed so by the leader. Therefore, some acceptors may “lag behind” with a $\Delta$-state that is smaller than what a safe quorum of acceptors have, not having received some messages from the leader, but they may not have inconsistent $\Delta$-states.

The refinement mapping has been checked by TLC using the consensus data type with four acceptors, three clients, and two consensus values, and with the generic data type with three acceptors, two clients, a unique command, and sequences of length smaller than or equal to 3.

The TLA+ specification of $ZLight$ can be found in appendix A.

#### Progress Guarantees

We can see that, to respond to a request, a client needs to receive acknowledgements from a safe quorum of acceptors and that a safe quorum of acceptors send their acknowledgements only after having received a message from the leader. Therefore to respond to a request the algorithm needs a correct safe quorum of acceptors and a correct leader. However, a client can panic at any time and then abort when it has received acknowledgements from a safe quorum of acceptors, without intervention of the leader. Therefore, if, eventually, a fast quorum of acceptors is correct for a long enough time, then a client will eventually abort its invocation even if the leader is faulty.

### 6.4.3 Progress Guarantees of $QZ$

Suppose that there eventually is a recovery quorum of acceptors which is correct for a long enough time. Since fast quorums can be bigger than recovery quorums, a $Quorum$ instance is not guaranteed to respond to requests. However, it is guaranteed to abort if the recovery quorum is correct for a long enough time. Assume that a $ZLight$ instance takes over $Quorum$ when it abort. Note that recovery quorums are at least as big as safe quorums. Therefore, if the leader of the $ZLight$ instance is correct then $ZLight$ will respond to the invocations if the recovery quorum is correct for a long enough time. If the leader is incorrect, then $ZLight$ will abort and a new instance of $ZLight$, with a different leader, can take over. Therefore we see that invocations eventually get responses when a recovery quorum of acceptors is correct for a long enough time. Strictly speaking, we would need to make some fairness assumptions about the appearance of $ZLight$ instances and about the rotation of leaders. Generalized Paxos has roughly the same progress guarantees as $QZ$.

## 6.5 Conclusion

We have applied speculative linearizability to build $QZ$, a robust linearizable algorithm in the message-passing computation model. $QZ$ is fault-tolerant and is an alternative to Generalized Paxos, a state of the art algorithm in the domain. Like Generalized Paxos, $QZ$ guarantees progress when a strict majority of the acceptors is eventually correct for a long enough time and $QZ$ can execute non-conflicting requests with a latency of two communication delays. However, being speculative linearizable, $QZ$ is easily extensible whereas Generalized Paxos is not. The $QZ$ algorithm also has the advantage that the relative sizes of fast and recovery quorums can be changed when changing mode instance.

We have also proposed two abstract specifications of safe and fast modes, which would simplify extending $QZ$ with new fast or safe modes.

The results of this chapter show that speculative linearizability can be used to build adaptive algorithms improving upon the state of the art in the field of fault-tolerant linearizable algorithms.

# 7 Applying Speculative Linearizability to Shared-Memory Consensus

In this chapter we present an adaptive, speculatively-linearizable, shared-memory consensus algorithm. Our consensus algorithm provides evidence that speculative linearizability can be used to build adaptive algorithms in the shared-memory model.

In shared memory, consensus cannot be implemented with atomic register [36]. However Luchangco et al. [58] presents an adaptive consensus algorithm which uses only atomic registers when clients do not contend for access to the shared memory and otherwise reverts to a consensus implementation that uses the compare-and-swap hardware instruction.

We propose an adaptive algorithm, inspired from Luchangco et al., composed of two speculatively linearizable modes $RegCons$ and $CASCons$. The mode $RegCons$ responds to invocations when clients do not contend. Otherwise $RegCons$ aborts and switches to $CASCons$, which uses the compare-and-swap hardware instruction to determine the consensus value.

The practical advantage of using only atomic registers in uncontended cases is not clear because modern processors execute a compare-and-swap instruction almost as fast as a load or a store with a memory fence [23]. Our adaptive consensus algorithm is therefore presented as a proof of concept that speculative linearizability can be applied to the shared memory model, but not as a new practical algorithm.

We assume that the clients only use the consensus implementation for a single invocation, even though our formal model of chapter 4 allows clients to submit new proposals after having received a response. In practice it would not make sense to reuse the consensus implementation once its output is decided.

The first consensus mode, $RegCons$, is presented, using pseudo code, in fig. 7.1. The $RegCons$ mode can only be used as a first mode, i.e., it has no init action.

The mode $RegCons$ uses a wait-free splitter algorithm. The splitter can be called by each client and takes no arguments; it guarantees that at most one client returns true, all others returning false. Moreover, it guarantees that, in the absence of contention, exactly one client returns true. The splitter algorithm can be implemented using only atomic registers as shown, using pseudo-code, in fig. 7.2. When discussing the pseudo code of figs. 7.1 and 7.2, we say that a client $c$ is at line $l$ when the statement at line $l$ is the *next* statement that $c$ will execute. Moreover, when a client executes a return statement of a response or a switch action (lines 8, 10, 17, 19, 23 of fig. 7.1, lines 7, 11, and 13 of fig. 7.2), then we consider that it stays, in the considered algorithm, at the corresponding line forever.

The following inductive invariant of the splitter implementation helps to understand its behavior. First add to the splitter a ghost variable $winner$, initialized to a special value "unset" and updated to the identity of the first client $p$ arriving at line 10 in a state where $X = p$. Note that when $p$ is at line 10, $p$ has not yet tested whether $X = p$ and might find it false when the test is performed, even thought $winner$ has been set to $p$. Observe that the following property is an inductive invariant: if $winner$ has been set, then for every client $p$, if $winner \neq p$ and $X = p$, then $p$ has not reached past line 8. When $winner$ changes from $unset$ to the identity of a process $p$, we have $X = p$ and $Y = true$. For another client $q \neq p$ to set $X$ to $q$, $q$ must be at line 5. Therefore it will find $Y = true$ at line 6 and return at line 7, never reaching past line 8.

Let us now examine the algorithm $RegCons$. Because at most one client returns true from the splitter, at most one client executes lines 14 to 19. Moreover, if this unique client $p$ returns $val_p$ at line 17, then it has seen, at line 16, $contention = false$. Therefore no client has executed line 22, which implies that no client switched and that every client will either return $val_p$ at line 8 or switch with $val_p$ at line 10 or 22. Therefore, once $p$ arrives at line 16 we can consider $val_p$ to be the chosen value, as in the refinement mapping below. We see that such an execution corresponds to an execution of $SLin$ in which $val_p$ is linearized an then every client aborts with or returns $val_p$.

Now assume that every process aborts. Because at most one client $p$ executes line 14 to 19, then every client aborts either with $\bot$, the initial value of $dState$, or with the value of $p$. Such an execution correspond to an execution of $SLin$ in which no request is linearized and every process aborts.

The argument elaborated in the last two paragraphs allows us to establish the correctness of $RegCons$ using the following refinement mapping.

**Theorem 7.1.** *The mode $RegCons$ is a speculatively linearizable first instance.*

*Proof.* Add to $RegCons$ the history variable $abortVals$, which is initially the empty set and is populated with the abort values produced by $RegCons$.

Define the function $f$ mappin a state $s$ of $RegCons$ the state $t$ of $SLin\ (Consensus)\ [1,2]$ as follows.

1. For every client $p$,

   (a) the pending request of $p$ in $t$ is the pending request of $p$ in $s$;

   (b) if $p$ is at lines 5, 8, or 17, then $status(t)[p]$ = "ready", if $p$ is at lines 10, 19, or 23, then $status(t)[p]$ = "aborted", and if $p$ is at any other line, then $status(t)[p]$ = "pending".

2. If there is a client $p$ at lines 16, 17 or 19, then $dState(t) = dState(s)$, else $dState(t) = \perp$.
3. The sets $abortVals$ are the same in $s$ and $t$;
4. The boolean $initialized(t)$ is true.
5. The set $initVals(t)$ is empty.

The function $f$ is a refinement mapping from $RegCons$ to $SLin(Consensus)[1,2]$. □

When the $RegCons$ mode aborts, it switches to the $CasCons$ mode, described in fig. 7.3. The $CasCons$ mode uses the compare-and-swap hardware instruction to choose a consensus value. The operation $CAS(dState, \perp, sval)$ atomically sets $dState$ to $sval$ if $dState = \perp$, and otherwise leaves $dState$ unchanged. It is easy to see that $CasCons$ implements $SLin(Consensus)[2,3]$.

We have shown, examining them in isolation from each other, that $RegCons$ and $CasCons$ are speculatively linearizable. Therefore, because $SLin$ is a modular property, we conclude that the adaptive algorithm whose first mode in $RegCons$ and whose second mode is $CasCons$ is a linearizable implementation of consensus.

This chapter has shown that speculative linearizability allows us to easily establish the correctness of the adaptive shared-memory algorithm $\{RegCons, CasCons\}$.

```
Algorithm $RCons_p$
Shared Δ-state $dState$, initially $\bot$
Shared boolean $decided$, initially $false$
Shared boolean $contention$, initially $false$
Function $Invoke_p^1(val)$:
if $decided = true$ then
    if $contention = false$ then
        $Return_p^1\,(dState)$
    else
        $Switch_p^2(val, dState)$
    end if
end if
if $Splitter(p) = true$ then
    $dState \leftarrow val$
    if $contention = false$ then
        $decided \leftarrow true$
        $Return_p^1\,(val)$
    else
        $Switch_p^2(val, \bot)$
    end if
else
    $contention \leftarrow true$
    $Switch_p^2(val, dState)$
end if
```

Figure 7.1 – The $RegCons$ Mode

```
Algorithm $Splitter$
Shared boolean $Y$, initially $false$
Shared process id $X$
Function $Splitter(p)$:
$X \leftarrow p$
if $Y = true$ then
    return $false$
end if
$Y \leftarrow true$
if $X = p$ then
    return $true$
else
    return $false$
end if
```

Figure 7.2 – The Splitter Algorithm

```
Algorithm $CasCons_p$
Shared Δ-state $dState$, initially $\bot$
Function $Switch_p^2\,(val, sval)$:
$CAS\,(dState, \bot, sval)$
$Response_p^2\,(dState)$
```

Figure 7.3 – The $CasCons$ Mode

# 8 Conclusion

We have seen that to be robust in practice, a distributed algorithm must have two important features.

1. A robust algorithm must adapt its strategy in response to change in the behavior of the system.
2. A robust algorithm must be easily extensible with new strategies, allowing incremental development.

Point 1 is a necessity in order to maintain the performance of the system despite unpredictable changes of behavior of its components. Point 2 is a necessity because the range of possible behavior of the system is not predictable a priori: new behaviors, mandating new strategies, are often discovered when the system is already in production.

With the example of State-Machine Replication, we have seen that the past, ad-hoc, approaches to building adaptive algorithms lead to impractical development costs and do not allow incremental development.

To tackle the problem, we have proposed a formal model of adaptive distributed algorithms which focuses on the switching mechanism of an adaptive algorithm, i.e., the problem of switching correctly, preserving safety and liveness, from one strategy to another. Strategies are modeled by families of I/O automata that we call modes. Using our model, we have defined the notion of modular property, which are the key enabler of practical adaptive algorithms.

A modular property is a correctness property that applies to a single mode, taken in isolation. A modular property guarantees that if each mode of an adaptive algorithm satisfies it, then the adaptive algorithm is correct. Modular properties therefore allow to reason modularly about adaptive algorithms, focusing on one mode at a time, incrementally. In contrast, the past, ad-hoc, approaches that we surveyed are not practical precisely because is is not possible to reason about modes independently.

To make the development of robust distributed-algorithms practical, we have proposed a modular property called Speculative Linearizability, forming, with our model of adaptive distributed algorithms, the Speculative Linearizability Framework. Speculation is a widely used approach to building efficient adaptive systems by employing optimistic strategies, at the cost of having to roll back overly optimistic computation. The Speculative Linearizability Framework allows building practical speculative algorithm which are linearizable implementations of data types.

To demonstrate the use of Speculative Linearizability, we have applied it to the problem of building fault-tolerant message-passing algorithms in the crash-stop fault model. Thanks to speculative linearizability, we have obtained $QZ$, an efficient and easily extensible algorithm solving the Generalized Consensus problem. The $QZ$ algorithm matches the state of the art in terms of latency and resilience to faults, notably optimizing the execution of commuting requests. However, state of the art algorithms are not easily extensible and would therefore become impractical as soon as a new behavior of the system makes their strategy inefficient.

We have also applied Speculative Linearizability to the problem of consensus in shared memory. We have proposed a consensus algorithm, inspired from [58], which uses only registers in uncontended cases. Although the practical implications of this algorithm are not clear, it shows that Speculative Linearizability can also be applied to shared memory and suggests investigating this area.

To avoid the notorious pitfalls of informal reasoning about distributed algorithms, we have formalized most of our work in TLA+, and we have proved an important result in Isabelle/HOL. The TLC model checker, analyzing TLA+ specifications, allows quick trial and error cycles that were instrumental in producing the results of this thesis.

We conclude by proposing directions for future research on the topic of practically building robust adaptive algorithms.

## 8.1 Future Work

### 8.1.1 Byzantine Faults in the Speculative Linearizability Framework

In the $QZ$ algorithm, a Byzantine client can make the system violate linearizability by sending wrong switch values when it changes mode. Making $QZ$ resilient to this type of fault would require cryptographic signatures to certify switch values, as done in the Byzantine fault-tolerant algorithms of the Abstract Framework [35].

However, the speculative linearizability framework cannot be used for Byzantine fault-tolerant algorithms because the interface of a mode instance does not contain any information about cryptographic keys, intercepted messages, etc. This information is necessary to soundly model Byzantine faults: in a real system, Byzantine processes could harvest cryptographic keys and signed messages in the first mode instance and then use them in the second instance, potentially compromising it. However this cannot be modeled in the speculative linearizability framework because the interface of a mode instance does not allow Byzantine processes to share information from one mode instance to the other.

To model Byzantine faults, the speculative linearizability framework would have to be modified by augmenting the interface of mode instances with actions modeling Byzantine processes acquiring knowledge about cryptographic keys and signed messages. The notion of modular property would have to be redefined to take into account the knowledge of Byzantine processes. A Byzantine speculative linearizability framework could be based on the ideas presented in [60] for modeling shared key communication systems using I/O automata, but the area remains to be explored.

### 8.1.2 Debugging Byzantine Fault-Tolerant Algorithms

A mechanically-checked proof should only be attempted when one has acquired a high degree of confidence in the truthfulness of the goal, but also about the usefulness of the goal: proving a statement of no practical interest is also a waste of time. Therefore we need prototyping tools that allow quickly exploring the problem space to find relevant statements that we would like to prove, and debugging tools to quickly find bugs and otherwise gain confidence that a statement is true before finally attempting its proof.

We have seen that the TLC model-checker allows fast prototyping and debugging in many cases, however it would not be efficient enough to handle Byzantine Fault-Tolerant algorithms. The state space and transition graph of such algorithms is especially large because a fraction of the processes, the Byzantine processes, are unrestricted in their actions. TLC was not able to check nontrivial properties of a Byzantine fault-tolerant version of Paxos in [47].

An interesting area of research would thus be to extend TLC or build another tool that allows fast prototyping of BFT algorithm. Symbolic reasoning technique would be required in order to analyze the arbitrary behavior of Byzantine processes, which results in too many possible cases to be analyzed by explicit state enumeration as employed by TLC.

### 8.1.3 Debugging Proofs at an Intermediate Level of Granularity

The construction of mechanically-checked proofs is made be much easier when prototyping and debugging tools allowed to check high-level properties before any proofs is attempted.

Let us draw an analogy with software testing. Software testing usually happens at three levels: unit testing, integration testing, and system testing. Unit tests exercise the functionality of small pieces of the system, i.e., individual functions or objects. Integration tests check that larger modules of the system behave correctly. System tests exercise the functionality of the full system end-to-end, from the point of view of its users.

When developing a theory in Isabelle/HOL, the “software” that the user would like to develop and test is the specifications *and* their proofs. The proofs are not an artifact of testing, but they are the subject of testing.

In software engineering, it is well-known that the cost of fixing an error grows rapidly as times advances. Therefore, as far as possible, one must not wait the completion of unit testing to start higher level tests.

In the subjective experience of the author, the testing tools available in Isabelle/HOL only allow unit tests. Therefore only a bottom-up approach to testing is possible and higher level errors are discovered late in the development process, at a high price. For example, after carefully decomposing the refinement proof between two I/O automata in dozens of smaller steps and sub-steps, we were able to test whether the individual steps were correct. However, we could not test whether the refinement was correct as a whole before decomposing it into small steps. When it turned out that one case was wrong and that the refinement or the I/O automata had to be changed, the meticulous decomposition had to be thrown away and the work had to be redone.

In contrast, the TLC model-checker excels at the highest level, i.e., end-to-end testing. For example we were able to test whether Quorum is correct by directly testing that it refines our specification of linearizability, without any intermediate steps. Moreover, this top-down approach did not lead to problems once we started refining our proofs and testing lower level refinement steps. Our experience therefore indicates that a top-down approach is much more efficient that a bottom-up approach.

However, we found neither TLC nor Nitpick to be adequate for testing the medium granularity structure of a proof. For example, it is hard to test whether the proof that $Lin'$ implements $Lin$ would be better carried out with a history variable in conjunction to a refinement mapping or with a forward simulation. Testing this fact would require writing a medium level proof skeleton for both cases and testing the individual steps of the skeletons. Such a test would reveal whether comparing the two skeletons is relevant. Without testing, one of the skeletons might just not be feasible even if it looks simpler. Discovering this fact halfway through the proof would be costly.

Exploring the testing of proofs specifically at an intermediate level of granularity would be an interesting research direction involving both technical challenges and human-computer interaction challenges.

### 8.1.4 Practical Applications of Speculative Linearizability in Shared-Memory

We have see in chapter 7 that Speculative Linearizability can be applied to build shared-memory algorithms. However we have only presented a proof of concept whose practical applications are unclear. The algorithm presented implements consensus in shared-memory and is composed of two modes. The first mode uses only atomic registers but is unable to make progress under contention, in which case the second mode takes over and reaches consensus using the compare-and-swap hardware instruction. On modern multiprocessors, the compare-and-swap instruction is roughly as fast as an atomic register access, i.e. a register access and a memory fence [23]. However the cost of atomic register access versus compare-and-swap may change in future multiprocessors and it may become advantageous to use only atomic register accesses when requests do not conflict. It would thus be interesting to investigate whether the idea underlying the shared-memory consensus algorithm generalize to the implementation of an arbitrary data type and whether avoiding the compare-and-swap instruction would make sense from the hardware point of view.

# Bibliography

[1] Michael ABD-EL-MALEK et al. “Fault-scalable Byzantine fault-tolerant services”. In: *SOSP*. Ed. by Andrew HERBERT and Kenneth P. BIRMAN. ACM, 2005, pp. 59–74. DOI: 10.1145/1095810.1095817.

[2] Dan ALISTARH et al. “On the cost of composing shared-memory algorithms”. In: *SPAA*. Ed. by Guy E. BLELLOCH and Maurice HERLIHY. ACM, 2012, pp. 298–307. DOI: 10.1145/2312005.2312057.

[3] Stuart F. ALLEN et al. “The Nuprl Open Logical Environment”. In: *CADE*. Ed. by David A. MCALLESTER. Vol. 1831. LNCS. Springer, 2000, pp. 170–176. DOI: 10.1007/10721959_12.

[4] Rajeev ALUR and Thomas A. HENZINGER. “Reactive Modules”. In: *Formal Methods in System Design* 15.1 (1999), pp. 7–48. DOI: 10.1023/A:1008739929481.

[5] Paul C. ATTIE and Nancy A. LYNCH. “Dynamic Input/Output Automata: A Formal Model for Dynamic Systems”. In: *CONCUR*. Ed. by Kim Guldstrand LARSEN and Mogens NIELSEN. Vol. 2154. LNCS. Springer, 2001, pp. 137–151. DOI: 10.1007/3-540-44685-0_10.

[6] Ananda BASU et al. “Rigorous Component-Based System Design Using the BIP Framework”. In: *IEEE Software* 28.3 (2011), pp. 41–48. DOI: 10.1109/MS.2011.27.

[7] Mark BICKFORD et al. “Proving Hybrid Protocols Correct”. In: *TPHOLs*. Ed. by Richard J. BOULTON and Paul B. JACKSON. Vol. 2152. LNCS. Springer, 2001, pp. 105–120. DOI: 10.1007/3-540-44755-5_9.

[8] Jasmin Christian BLANCHETTE, Sascha BÖHME, and Lawrence C. PAULSON. “Extending Sledgehammer with SMT Solvers”. In: *J. Autom. Reasoning* 51.1 (2013), pp. 109–128. DOI: 10.1007/s10817-013-9278-5.

[9] Jasmin Christian BLANCHETTE and Tobias NIPKOW. “Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder”. In: *ITP*. Ed. by Matt KAUFMANN and Lawrence C. PAULSON. Vol. 6172. LNCS. Springer, 2010, pp. 131–146. DOI: 10.1007/978-3-642-14052-5_11.

[10] Egon BÖRGER and Robert F STÄRK. *Abstract state machines: a method for high-level system design and analysis*. Vol. 14. Springer Heidelberg, 2003.

[11] Marius BOZGA et al. “Modeling Dynamic Architectures Using Dy-BIP”. In: *Software Composition*. Ed. by Thomas GSCHWIND et al. Vol. 7306. LNCS. Springer, 2012, pp. 1–16. DOI: 10.1007/978-3-642-30564-1_1.

[12] Lásaro J. CAMARGOS, Rodrigo SCHMIDT, and Fernando PEDONE. “Multicoordinated Paxos”. In: *PODC*. Ed. by Indranil GUPTA and Roger WATTENHOFER. ACM, 2007, pp. 316–317. DOI: 10.1145/1281100.1281150.

[13] Miguel CASTRO and Barbara LISKOV. *A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm*. Technical Memo MIT-LCS-TM-590. MIT, 1999.

[14] Miguel CASTRO and Barbara LISKOV. “Practical byzantine fault tolerance and proactive recovery”. In: *ACM Trans. Comput. Syst.* 20.4 (2002), pp. 398–461. DOI: 10.1145/571637.571640.

[15] Ilwoo CHANG, Matti A. HILTUNEN, and Richard D. SCHLICHTING. “Affordable Fault Tolerance Through Adaptation”. In: *IPPS/SPDP Workshops*. 1998, pp. 585–603. DOI: 10.1007/3-540-64359-1_730.

[16] Bernadette CHARRON-BOST and André SCHIPER. “The Heard-Of model: computing in distributed systems with benign faults”. In: *Distributed Computing* 22.1 (2009), pp. 49–71. DOI: 10.1007/s00446-009-0084-6.

[17] Wen-Ke CHEN, Matti A. HILTUNEN, and Richard D. SCHLICHTING. “Constructing Adaptive Software in Distributed Systems”. In: *ICDCS*. 2001, pp. 635–643. DOI: 10.1109/ICDSC.2001.918994.

[18] A. CIMATTI et al. “NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking”. In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, 2002.

[19] Allen CLEMENT et al. “Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults”. In: *NSDI*. Ed. by Jennifer REXFORD and Emin Gün SIRER. USENIX Association, 2009, pp. 153–168.

[20] RL CONSTABLE et al. *Implementing mathematics*. Citeseer, 1986.

[21] Denis COUSINEAU et al. “TLA+ Proofs”. In: *CoRR*. LNCS abs/1208.5933 (2012). Ed. by Dimitra GIANNAKOPOULOU and Dominique MÉRY, pp. 147–154. DOI: 10.1007/978-3-642-32759-9_14.

[22] James A. COWLING et al. “HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance”. In: *OSDI*. Ed. by Brian N. BERSHAD and Jeffrey C. MOGUL. USENIX Association, 2006, pp. 177–190.

[23] Tudor DAVID, Rachid GUERRAOUI, and Vasileios TRIGONAKIS. “Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask”. In: *SOSP*. Ed. by Michael KAMINSKY and Mike DAHLIN. ACM, 2013, pp. 33–48. DOI: 10.1145/2517349.2522714.

[24] Tzilla ELRAD and Nissim FRANCEZ. “Decomposition of Distributed Programs into Communication-Closed Layers”. In: *Sci. Comput. Program.* 2.3 (1982), pp. 155–173. DOI: 10.1016/0167-6423(83)90013-8.

[25] E. Allen EMERSON and Vineet KAHLON. “Model Checking Large-Scale and Parameterized Resource Allocation Systems”. In: *TACAS*. Ed. by Joost-Pieter KATOEN and Perdita STEVENS. Vol. 2280. LNCS. Springer, 2002, pp. 251–265. DOI: 10.1007/3-540-46002-0_18.

[26] E. Allen EMERSON and Vineet KAHLON. “Reducing Model Checking of the Many to the Few”. In: *CADE*. Ed. by David A. MCALLESTER. Vol. 1831. LNCS. Springer, 2000, pp. 236–254. DOI: 10.1007/10721959_19.

[27] E. Allen EMERSON and Kedar S. NAMJOSHI. “Reasoning about Rings”. In: *POPL*. Ed. by Ron K. CYTRON and Peter LEE. ACM Press, 1995, pp. 85–94. DOI: 10.1145/199448.199468.

[28] Ivana FILIPOVIC et al. “Abstraction for concurrent objects”. In: *Theor. Comput. Sci.* 411.51-52 (2010), pp. 4379–4398. DOI: 10.1016/j.tcs.2010.09.021.

[29] Eli GAFNI and Leslie LAMPORT. “Disk Paxos”. In: *Distributed Computing* 16.1 (2003), pp. 1–20. DOI: 10.1007/s00446-002-0070-8.

[30] Stephen J. GARLAND and John V. GUTTAG. “LP: The Larch Prover”. In: *CADE*. Ed. by Ewing L. LUSK and Ross A. OVERBEEK. Vol. 310. LNCS. Springer, 1988, pp. 748–749. DOI: 10.1007/BFb0012879.

[31] Chryssis GEORGIOU et al. “Automated implementation of complex distributed algorithms specified in the IOA language”. In: *STTT* 11.2 (2009), pp. 153–171. DOI: 10.1007/s10009-008-0097-7.

[32] Rachid GUERRAOUI, Viktor KUNCAK, and Giuliano LOSA. “Abortable Linearizable Modules”. In: *The Archive of Formal Proofs*. Ed. by Gerwin KLEIN, Tobias NIPKOW, and Lawrence PAULSON. Formal proof development. http://afp.sf.net/entries/Abortable_Linearizable_Modules.shtml, 2012.

[33] Rachid GUERRAOUI, Viktor KUNCAK, and Giuliano LOSA. “Speculative Linearizability”. In: *PLDI*. Ed. by Jan VITEK, Haibo LIN, and Frank TIP. ACM, 2012, pp. 55–66. DOI: 10.1145/2254064.2254072.

[34] Rachid GUERRAOUI and Marko VUKOLIC. “Refined quorum systems”. In: *Distributed Computing* 23.1 (2010), pp. 1–42. DOI: 10.1007/s00446-010-0103-7.

[35] Rachid GUERRAOUI et al. “The next 700 BFT protocols”. In: *EuroSys*. Ed. by Christine MORIN and Gilles MULLER. ACM, 2010, pp. 363–376. DOI: 10.1145/1755913.1755950.

[36] Maurice HERLIHY. “Wait-Free Synchronization”. In: *ACM Trans. Program. Lang. Syst.* 13.1 (1991), pp. 124–149. DOI: 10.1145/114005.102808.

[37] Maurice HERLIHY and Jeannette M. WING. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972.

[38] Matti A. HILTUNEN and Richard D. SCHLICHTING. “A Model for Adaptive Fault-Tolerant Systems”. In: *EDCC*. Ed. by Klaus ECHTLE, Dieter K. HAMMER, and David POWELL. Vol. 852. LNCS. Springer, 1994, pp. 3–20. DOI: 10.1007/3-540-58426-9_121.

[39] C. A. R. HOARE. “Communicating Sequential Processes”. In: *Commun. ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585.

[40] Gerard J. HOLZMANN. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004, pp. I–XII, 1–596. ISBN: 978-0-321-22862-8.

[41] Michel HURFIN, Izabela MOISE, and Jean-Pierre Le NARZUL. "An Adaptive Fast Paxos for Making Quick Everlasting Decisions". In: *AINA.* IEEE Computer Society, 2011, pp. 208–215. DOI: 10.1109/AINA.2011.73.

[42] *IOA Language and Toolset (web page).* https://groups.csail.mit.edu/tds/ioa/. Accessed: 2013-10-18. 2003.

[43] Mauro JASKELIOFF and Stephan MERZ. "Proving the Correctness of Disk Paxos". In: *The Archive of Formal Proofs.* Ed. by Gerwin KLEIN, Tobias NIPKOW, and Lawrence PAULSON. Formal proof development. http://afp.sf.net/entries/DiskPaxos.shtml, 2005.

[44] Prasad JAYANTI. "Adaptive and efficient abortable mutual exclusion". In: *PODC.* Ed. by Elizabeth BOROWSKY and Sergio RAJSBAUM. ACM, 2003, pp. 295–304. DOI: 10.1145/872035.872079.

[45] Florian KAMMÜLLER, Markus WENZEL, and Lawrence C. PAULSON. "Locales - A Sectioning Concept for Isabelle". In: *TPHOLs.* Ed. by Yves BERTOT et al. Vol. 1690. LNCS. Springer, 1999, pp. 149–166. DOI: 10.1007/3-540-48256-3_11.

[46] Ramakrishna KOTLA et al. "Zyzzyva: Speculative Byzantine fault tolerance". In: *ACM Trans. Comput. Syst.* 27.4 (2009). DOI: 10.1145/1658357.1658358.

[47] Leslie LAMPORT. "Byzantizing Paxos by Refinement". In: *DISC.* Ed. by David PELEG. Vol. 6950. LNCS. Springer, 2011, pp. 211–224. DOI: 10.1007/978-3-642-24100-0_22.

[48] Leslie LAMPORT. "Fast Paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103. DOI: 10.1007/s00446-006-0005-x.

[49] Leslie LAMPORT. *Generalized Consensus and Paxos.* https://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#generalized. Accessed: 2013-10-18. 2005.

[50] Leslie LAMPORT. "Lower bounds for asynchronous consensus". In: *Distributed Computing* 19.2 (2006), pp. 104–125. DOI: 10.1007/s00446-006-0155-x.

[51] Leslie LAMPORT. "On Interprocess Communication. Part I: Basic Formalism". In: *Distributed Computing* 1.2 (1986), pp. 77–85. DOI: 10.1007/BF01786227.

[52] Leslie LAMPORT. "On Interprocess Communication. Part II: Algorithms". In: *Distributed Computing* 1.2 (1986), pp. 86–101. DOI: 10.1007/BF01786228.

[53] Leslie LAMPORT. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002. ISBN: 0-3211-4306-X.

[54] Leslie LAMPORT. "The Implementation of Reliable Distributed Multiprocess Systems". In: *Computer Networks* 2 (1978), pp. 95–114. DOI: 10.1016/0376-5075(78)90045-4.

[55] Leslie LAMPORT, Dahlia MALKHI, and Lidong ZHOU. "Vertical paxos and primary-backup replication". In: *PODC.* Ed. by Srikanta TIRTHAPURA and Lorenzo ALVISI. ACM, 2009, pp. 312–313. DOI: 10.1145/1582716.1582783.

[56] Leslie LAMPORT and Mike MASSA. “Cheap Paxos”. In: *DSN*. IEEE Computer Society, 2004, pp. 307–314. DOI: 10.1109/DSN.2004.1311900.

[57] Barbara LISKOV and Stephen N. ZILLES. “Programming with Abstract Data Types”. In: *SIGPLAN Notices* 9.4 (1974), pp. 50–59.

[58] Victor LUCHANGCO, Mark MOIR, and Nir SHAVIT. “On the Uncontended Complexity of Consensus”. In: *DISC*. Ed. by Faith Ellen FICH. Vol. 2848. LNCS. Springer, 2003, pp. 45–59. DOI: 10.1007/978-3-540-39989-6_4.

[59] Nancy A. LYNCH. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN: 1-55860-348-4.

[60] Nancy A. LYNCH. “I/O Automaton Models and Proofs for Shared-Key Communication Systems”. In: *CSFW*. IEEE Computer Society, 1999, pp. 14–29. DOI: 10.1109/CSFW.1999.779759.

[61] Nancy A. LYNCH and Mark R. TUTTLE. “An introduction to input/output automata”. In: *CWI Quarterly* 2 (1989), pp. 219–246.

[62] Nancy A. LYNCH and Frits W. VAANDRAGER. “Forward and Backward Simulations: I. Untimed Systems”. In: *Inf. Comput.* 121.2 (1995), pp. 214–233. DOI: 10.1006/inco.1995.1134.

[63] Yanhua MAO, Flavio Paiva JUNQUEIRA, and Keith MARZULLO. “Mencius: Building Efficient Replicated State Machine for WANs”. In: *OSDI*. Ed. by Richard DRAVES and Robbert van RENESSE. USENIX Association, 2008, pp. 369–384.

[64] Parisa Jalili MARANDI, Marco PRIMI, and Fernando PEDONE. “Multi-Ring Paxos”. In: *DSN*. Ed. by Robert S. SWARZ, Philip KOOPMAN, and Michel CUKIER. IEEE Computer Society, 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263916.

[65] Parisa Jalili MARANDI et al. “Ring Paxos: A high-throughput atomic broadcast protocol”. In: *DSN*. IEEE, 2010, pp. 527–536. DOI: 10.1109/DSN.2010.5544272.

[66] Jean-Philippe MARTIN and Lorenzo ALVISI. “Fast Byzantine Consensus”. In: *IEEE Trans. Dependable Sec. Comput.* 3.3 (2006), pp. 202–215. DOI: 10.1109/TDSC.2006.35.

[67] Antoni W. MAZURKIEWICZ. “Semantics of concurrent systems: a modular fixed-point trace approach”. In: *European Workshop on Applications and Theory in Petri Nets*. 1984, pp. 353–375.

[68] Philip K. MCKINLEY et al. “Composing Adaptive Software”. In: *IEEE Computer* 37.7 (2004), pp. 56–64. DOI: 10.1109/MC.2004.48.

[69] Stephan MERZ. “The specification language TLA+”. In: *Logics of specification languages*. Springer, 2008, pp. 401–451.

[70] Robin MILNER. “Bigraphical Reactive Systems”. In: *CONCUR*. Ed. by Kim Guldstrand LARSEN and Mogens NIELSEN. Vol. 2154. LNCS. Springer, 2001, pp. 16–35. DOI: 10.1007/3-540-44685-0_2.

[71] Robin MILNER, Joachim PARROW, and David WALKER. “A Calculus of Mobile Processes, I”. In: *Inf. Comput.* 100.1 (1992), pp. 1–40.

[72] Robin MILNER, Joachim PARROW, and David WALKER. “A Calculus of Mobile Processes, II”. In: *Inf. Comput.* 100.1 (1992), pp. 41–77.

[73] Iulian MORARU, David G. ANDERSEN, and Michael KAMINSKY. “There is more consensus in Egalitarian parliaments”. In: *SOSP*. Ed. by Michael KAMINSKY and Mike DAHLIN. ACM, 2013, pp. 358–372. DOI: 10.1145/2517349.2517350.

[74] Olaf MÜLLER. “I/O Automata and Beyond: Temporal Logic and Abstraction in Isabelle”. In: *TPHOLs*. 1998, pp. 331–348.

[75] Olaf MÜLLER and Tobias NIPKOW. “Combining Model Checking and Deduction for I/O-Automata”. In: *TACAS*. Ed. by Ed BRINKSMA et al. Vol. 1019. LNCS. Springer, 1995, pp. 1–16. DOI: 10.1007/3-540-60630-0_1.

[76] Tobias NIPKOW, Lawrence C. PAULSON, and Markus WENZEL. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7.

[77] Peyman OREIZY, Nenad MEDVIDOVIC, and Richard N. TAYLOR. “Runtime software adaptation: framework, approaches, and styles”. In: *ICSE Companion*. Ed. by Wilhelm SCHÄFER, Matthew B. DWYER, and Volker GRUHN. ACM, 2008, pp. 899–910. DOI: 10.1145/1370175.1370181.

[78] Peyman OREIZY et al. “An architecture-based approach to self-adaptive software”. In: *Intelligent Systems and Their Applications, IEEE* 14.3 (1999), pp. 54–62.

[79] Lawrence C. PAULSON. “A Generic Tableau Prover and its Integration with Isabelle”. In: *J. UCS* 5.3 (1999), pp. 73–87.

[80] Lawrence C. PAULSON. “Isabelle: The Next 700 Theorem Provers”. In: *CoRR* cs.LO/9301106 (1993).

[81] Fernando PEDONE. “Boosting System Performance with Optimistic Distributed Protocols”. In: *IEEE Computer* 34.12 (2001), pp. 80–86. DOI: 10.1109/2.970581.

[82] C. A. PETRI. “Fundamentals of a Theory of Asynchronous Information Flow”. In: *IFIP Congress*. 1962, pp. 386–390.

[83] Robbert van RENESSE et al. “Building Adaptive Systems Using Ensemble”. In: *Softw., Pract. Exper.* 28.9 (1998), pp. 963–979. DOI: 10.1002/(SICI)1097-024X(19980725)28:9<963::AID-SPE179>3.0.CO;2-9.

[84] Liliana ROSA et al. “Self-Management of Adaptable Component-Based Applications”. In: *IEEE Trans. Software Eng.* 39.3 (2013), pp. 403–421. DOI: 10.1109/TSE.2012.29.

[85] Olivier RÜTTI and André SCHIPER. “A predicate-based approach to dynamic protocol update in group communication”. In: *IPDPS*. IEEE, 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536238.

[86] Olivier RÜTTI, Pawel T. WOJCIECHOWSKI, and André SCHIPER. “Structural and algorithmic issues of dynamic protocol update”. In: *IPDPS*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639369.

[87] Fred B. SCHNEIDER. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319. DOI: 10.1145/98163.98167.

[88] Atul SINGH et al. “BFT Protocols Under Fire”. In: *NSDI*. Ed. by Jon CROWCROFT and Michael DAHLIN. USENIX Association, 2008, pp. 189–204.

[89] Dmitriy TRAYTEL, Andrei POPESCU, and Jasmin Christian BLANCHETTE. “Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving”. In: *LICS*. IEEE, 2012, pp. 596–605. DOI: 10.1109/LICS.2012.75.

[90] Wei WEI et al. “Fast Mencius: Mencius with low commit latency”. In: *INFOCOM*. IEEE, 2013, pp. 881–889. DOI: 10.1109/INFCOM.2013.6566876.

[91] Markus WENZEL. “Isar - A Generic Interpretative Approach to Readable Formal Proof Documents”. In: *TPHOLs*. Ed. by Yves BERTOT et al. Vol. 1690. LNCS. Springer, 1999, pp. 167–184. DOI: 10.1007/3-540-48256-3_12.

[92] Toh Ne WIN et al. “Using simulated execution in verifying distributed algorithms”. In: *STTT* 6.1 (2004), pp. 67–76. DOI: 10.1007/s10009-003-0126-5.

[93] Pawel T. WOJCIECHOWSKI and Olivier RÜTTI. “On Correctness of Dynamic Protocol Update”. In: *FMOODS*. Ed. by Martin STEFFEN and Gianluigi ZAVATTARO. Vol. 3535. LNCS. Springer, 2005, pp. 275–289. DOI: 10.1007/11494881_18.

[94] Yuan YU, Panagiotis MANOLIOS, and Leslie LAMPORT. “Model Checking TLA$^+$ Specifications”. In: *CHARME*. Ed. by Laurence PIERRE and Thomas KROPF. Vol. 1703. LNCS. Springer, 1999, pp. 54–66. DOI: 10.1007/3-540-48153-2_6.

# A TLA+ Specifications

In this appendix we include all the TLA+ specifications of the algorithms presented in the thesis. The specifications and their properties, including the composition theorem, have all been exhaustively model checked, for small system sizes and with the three different data types, with the TLC model checker.

```
──────────────────── MODULE Library ────────────────────
EXTENDS $Sequences, Naturals, FiniteSets$

$Last(s) \triangleq s[Len(s)]$

$Some(S) \triangleq$ CHOOSE $e \in S :$ TRUE

$Min(i, j) \triangleq$ IF $i < j$ THEN $i$ ELSE $j$
$Max(S, LessEq(\_, \_)) \triangleq$ CHOOSE $e \in S : \forall e1 \in S : LessEq(e1, e)$

$Image(f) \triangleq \{f[x] : x \in$ DOMAIN $f\}$

Sequences with no duplicates:

RECURSIVE $NoDupRec(\_, \_)$
$NoDupRec(es, seen) \triangleq$
    IF $es = \langle\rangle$
     THEN TRUE
     ELSE
        IF $es[1] \in seen$
         THEN FALSE
         ELSE $NoDupRec(Tail(es), seen \cup \{es[1]\})$
$NoDup(es) \triangleq$
  $NoDupRec(es, \{\})$
$NoDupSeq(E) \triangleq$
  $\{es \in Seq(E) : NoDup(es)\}$

Removing duplicates from a sequence:

RECURSIVE $RemDupRec(\_, \_)$
$RemDupRec(es, seen) \triangleq$
  IF $es = \langle\rangle$
   THEN $\langle\rangle$
   ELSE
    IF $es[1] \in seen$
     THEN $RemDupRec(Tail(es), seen)$
     ELSE $\langle es[1]\rangle \circ RemDupRec(Tail(es), seen \cup \{es[1]\})$
$RemDup(es) \triangleq RemDupRec(es, \{\})$

Sequence prefixes:

$Prefix(s1, s2) \triangleq$
    $\land\ Len(s1) \leq Len(s2)$
    $\land\ \forall i \in$ DOMAIN $s1 : s1[i] = s2[i]$

The longest common prefix of two sequences:

RECURSIVE $LongestCommonPrefixLenRec(\_, \_, \_)$
```

```
$LongestCommonPrefixLenRec(S, n, e1) \triangleq$
   IF $S = \{\}$
    THEN $0$
    ELSE
       IF $\land \forall e \in S : Len(e) \; \geq n + 1$
           $\land \forall e \in S : e[n + 1] = e1[n + 1]$
        THEN $LongestCommonPrefixLenRec(S, n + 1, e1)$
        ELSE $n$

$LongestCommonPrefixLenSet(S) \triangleq LongestCommonPrefixLenRec(S, 0, Some(S))$

$LongestCommonPrefix(S) \triangleq$
   LET $n \triangleq LongestCommonPrefixLenSet(S)$
   IN  IF $n = 0$
        THEN $\langle\rangle$
        ELSE $[i \in 1 \,..\, LongestCommonPrefixLenSet(S) \mapsto Some(S)[i]]$
```

## A.1 Speculative Linearizability

```
MODULE $RDR$

Specification of Recoverable Data-Type Representations

EXTENDS $Sequences, Naturals, FiniteSets, Library$

CONSTANTS $S, C, O, P, \_ \bullet \_, Output(\_, \_), Bot$
  For the efficiency of model checking, allow substitution of star, $GLB$,
  and $Contains$. The properties of the constants below are asserted in
  ASSUME statements.
CONSTANTS $\_ \star \_, GLB(\_), Contains(\_, \_), \_ \preceq \_$

  Requests:
$Req \triangleq P \times C$

  Types of $\bullet$ and $Output$:
$TypeOk \triangleq$
    $\land \forall s \in S, c \in Req : s \bullet c \in S$
    $\land \forall s \in S, c \in Req : Output(s, c) \in O$
ASSUME $TypeOk$

  Execute a sequence of requests:
RECURSIVE $Star(\_, \_, \_)$
$Star(s, rs, i) \triangleq$
    IF $Len(rs) < i$ THEN $s$
     ELSE LET $s2 \triangleq s \bullet rs[i]$IN $Star(s2, rs, i + 1)$
  Ensures that $\star$ and $Star$ match.
ASSUME $\forall s \in S, rs \in Seq(Req) : s \star rs = Star(s, rs, 1)$


  Idempotence property of data types:
$Idem1 \triangleq \forall s \in S : \forall r \in Req : \forall rs \in Seq(Req) : r \in Image(rs) \Rightarrow s \star rs = s \star Append(rs, r)$
$Idem2 \triangleq \forall s \in S : \forall o \in O : \forall p, q \in P : \forall c1, c2 \in C :$
   LET $r1 \triangleq \langle p, c1 \rangle$
       $r2 \triangleq \langle q, c2 \rangle$
   IN
      $Output(s, r1) = o \land p \neq q$
         $\Rightarrow$ LET $s2 \triangleq (s \bullet r1) \bullet r2$
            IN $Output(s2, r1) = o$
$Idem \triangleq Idem1 \land Idem2$
ASSUME $Idem$

  The partial order:
$PrecEq(s1, s2) \triangleq$
    $\lor s1 = s2$
    $\lor \exists rs \in Seq(Req) : s2 = s1 \star rs$
ASSUME $\forall s1, s2 \in S : (s1 \preceq s2) = PrecEq(s1, s2)$
```

```
Antisymmetry of RDRs
$AntiSym \triangleq \forall s1, s2 \in S : s1 \preceq s2 \land s2 \preceq s1 \Rightarrow s1 = s2$
ASSUME $AntiSym$

Greatest lower bounds:
$IsLB(s, s1, s2) \triangleq s \preceq s1 \land s \preceq s2$
$IsGLB(s, s1, s2) \triangleq$
    $\land IsLB(s, s1, s2)$
    $\land \forall s3 \in S : s \neq s3 \land IsLB(s3, s1, s2) \Rightarrow s3 \preceq s$
Semi lattice property of RDRs:
$s1 \sqcap s2 \triangleq$ CHOOSE $s \in S : IsGLB(s, s1, s2)$

$GLBExists \triangleq \forall s1, s2 \in S : IsGLB(s1 \sqcap s2, s1, s2)$
ASSUME $GLBExists$

GLB of a set of states:
RECURSIVE $GLB1(\_)$
$GLB1(ss) \triangleq$
        LET $s \triangleq$ CHOOSE $s \in ss$ : TRUE
        IN
            IF $Cardinality(ss) = 1$
             THEN $s$
             ELSE $(s \sqcap GLB1(ss \setminus \{s\}))$
ASSUME $\forall ss \in$ SUBSET $S \setminus \{\{\}\} : GLB1(ss) = GLB(ss)$

The consistency property of RDRs:
$Consistency \triangleq \forall s1, s2 \in S, rs \in Seq(Req) :$
    LET $s3 \triangleq s1 \star rs$
    IN  $\land s1 \preceq s2$
        $\land s2 \preceq s3$
        $\Rightarrow \exists rs1, rs2 \in Seq(Req) :$
             $\land s2 = s1 \star rs1$
             $\land s3 = s2 \star rs2$
             $\land Image(rs1) \subseteq Image(rs)$
             $\land Image(rs2) \subseteq Image(rs)$
ASSUME $Consistency$

Checking whether an RDR contains a given request:
$Contains1(s, r) \triangleq \exists rs \in Seq(Req) : r \in Image(rs) \land s = Bot \star rs$
ASSUME $\forall s \in S, r \in Req : Contains(s, r) = Contains1(s, r)$
```

```
MODULE TestAndSet

CONSTANTS $P$
$C \triangleq \{$“ts”$\}$
$O \triangleq \{$“Won”, “Lost”$\}$
$S \triangleq \{P\} \cup P$
$Bot \triangleq P$
$s \bullet r \triangleq$
 IF $s = P$ THEN $r[1]$ ELSE $s$
$Output(s, r) \triangleq$
  IF $s = P$ THEN “Won” ELSE IF $r[1] = s$ THEN “Won” ELSE “Lost”
$s1 \preceq s2 \triangleq$
   $\vee\, s1 = s2 \vee s1 = P$
$s \star rs \triangleq$
 IF $rs = \langle\rangle \vee s \neq Bot$ THEN $s$
  ELSE $rs[1][1]$
$GLB(ss) \triangleq$
  IF $ss = \{\}$ THEN $Bot$
   ELSE
    IF $\exists\, s1, s2 \in ss : s1 \neq s2$
     THEN $Bot$
     ELSE CHOOSE $s \in ss$ : TRUE
$Contains(s, r) \triangleq$
  IF $s = P$ THEN FALSE ELSE TRUE
```

```
───────────────── MODULE $Consensus$ ─────────────────
EXTENDS $Sequences$

CONSTANTS $P, V$
$C \triangleq V$
$O \triangleq V$
$S \triangleq \{V\} \cup V$
$Bot \triangleq V$
$s \bullet r \triangleq$
 IF $s = V$ THEN $r[2]$ ELSE $s$
$Output(s, r) \triangleq$
  IF $s = V$ THEN $r[2]$ ELSE $s$
$s1 \preceq s2 \triangleq$
   $\lor\ s1 = s2 \lor s1 = V$
$s \star rs \triangleq$
 IF $rs = \langle\rangle \lor s \neq Bot$ THEN $s$
  ELSE $rs[1][2]$
$GLB(ss) \triangleq$
  IF $ss = \{\}$ THEN $Bot$
   ELSE
    IF $\exists\, s1, s2 \in ss : s1 \neq s2$
     THEN $Bot$
     ELSE CHOOSE $s \in ss$ : TRUE
$Contains(s, r) \triangleq$
  IF $s = V$ THEN FALSE ELSE TRUE

──────────────────────────────────────────────────────
```

```
┌──────────────────────── MODULE Generic ────────────────────────┐

EXTENDS Library

CONSTANTS P, C
O ≜ Seq(P × C)
S ≜ {rs ∈ Seq(P × C) : NoDup(rs, {})}
Bot ≜ ⟨⟩
s • r            ≜ IF r ∈ Image(s) THEN s ELSE Append(s, r)
Output(s, r) ≜ IF r ∈ Image(s) THEN Truncate(r, s) ELSE Append(s, r)
s1 ⪯ s2 ≜
  Prefix(s1, s2)
s ⋆ rs ≜ s ∘ RemDup(rs)
GLB(ss) ≜ LongestCommonPrefix(ss)
Contains(s, r) ≜ r ∈ Image(s)

└─────────────────────────────────────────────────────────────────┘
```

```
──────────────────────── MODULE LinInterface ────────────────────────
EXTENDS Library

CONSTANTS P, C, S, O
VARIABLE interface

InvInterfaceType ≜ [P → [cmd : C, flag : BOOLEAN ]]
RespInterfaceType ≜ [P → [output : O, flag : BOOLEAN ]]

InterfaceType ≜ [
    inv : InvInterfaceType,
    resp : RespInterfaceType]

InvInterfaceInit ≜ [p ∈ P ↦ [
    cmd ↦ Some(C),
    flag ↦ Some(BOOLEAN )]]

RespInterfaceInit ≜ [p ∈ P ↦ [
    output ↦ Some(O),
    flag ↦ Some(BOOLEAN )]]

InterfaceInit ≜ [
    inv ↦ InvInterfaceInit,
    resp ↦ RespInterfaceInit]

Invoke(p, cmd) ≜
    interface′ = [interface EXCEPT !.inv = [@ EXCEPT ![p] = [
        cmd ↦ cmd,
        flag ↦ ¬@.flag]]]

Response(p, o) ≜
    interface′ = [interface EXCEPT !.resp = [@ EXCEPT ![p] = [
        output ↦ o,
        flag ↦ ¬@.flag]]]
─────────────────────────────────────────────────────────────────────
```

```
┌──────────────── MODULE Linearizability ────────────────┐
EXTENDS RDR

VARIABLES
    status, pending, dState, nxtOut, interface

INSTANCE LinInterface

vars ≜ ⟨status, pending, dState, nxtOut, interface⟩

Label ≜ {"ready", "committed", "pending"}  The status of a process.

TypeInvariant ≜
    ∀ p ∈ P :
        ∧ status[p] ∈ Label
        ∧ pending[p] ∈ C
        ∧ nxtOut[p] = O
    ∧ dState ∈ S

 Invocation by process p:
Inv(p) ≜ ∃ c ∈ C :
      ∧ status[p] = "ready"
      ∧ status′ = [status EXCEPT ![p] = "pending"]
      ∧ pending′ = [pending EXCEPT ![p] = ⟨p, c⟩]
      ∧ Invoke(p, c)
      ∧ UNCHANGED ⟨dState, nxtOut⟩

 Response by process p:
Resp(p) ≜
      ∧  status[p] = "committed"
      ∧  status′ = [status EXCEPT ![p] = "ready"]
      ∧  Response(p, nxtOut[p])
      ∧  UNCHANGED ⟨dState, pending, nxtOut⟩

 Linearize one pending request.
Lin ≜
    ∧ ∃ p ∈ P :
        ∧  status[p] = "pending"
        ∧  status′  = [status EXCEPT ![p] = "committed"]
        ∧  dState′  = dState • pending[p]
        ∧  nxtOut′ = [nxtOut EXCEPT ![p] = Output(dState, pending[p])]
    ∧ UNCHANGED ⟨pending, interface⟩

Init ≜
    ∧ status  = [p ∈ P ↦ "ready"]
    ∧ dState = Bot
    ∧ pending = [p ∈ P ↦ Some(Req)]
```

$\quad\land\ nxtOut = [p \in P \mapsto Some(O)]$
$\quad\land\ interface = InterfaceInit$

$Next \ \triangleq\ Lin \lor (\exists\, p \in P : Inv(p) \lor Resp(p))$

$Spec \ \triangleq\ Init \land \Box[Next]_{vars}$

```
──────────────────────── MODULE SpecLinInterface ────────────────────────
EXTENDS Library

CONSTANTS P, C, S, O

VARIABLE interface

LI ≜ INSTANCE LinInterface

SwitchInterfaceType ≜ [P → [cmd : C, sval : S, flag : BOOLEAN ]]

InterfaceType ≜ [
    init :  SwitchInterfaceType,
    inv : LI!InvInterfaceType,
    resp : LI!RespInterfaceType,
    abort : SwitchInterfaceType]

SwitchInterfaceInit ≜ [p ∈ P ↦ [
    cmd ↦ Some(C),
    sval ↦ Some(S),
    flag ↦ Some(BOOLEAN )]]

InterfaceInit ≜ [
    init ↦ SwitchInterfaceInit,
    inv ↦ LI!InvInterfaceInit,
    resp ↦ LI!RespInterfaceInit,
    abort ↦ SwitchInterfaceInit]

Invoke(p, cmd) ≜ LI!Invoke(p, cmd)

Response(p, o) ≜ LI!Response(p, o)

Initialize(p, cmd, sv) ≜
    interface′ = [interface EXCEPT !.init = [@ EXCEPT ![p] = [
        cmd ↦ cmd,
        sval ↦ sv,
        flag ↦ ¬@.flag]]]

Abort(p, cmd, sv) ≜
    interface′ = [interface EXCEPT !.abort = [@ EXCEPT ![p] = [
        cmd ↦ cmd,
        sval ↦ sv,
        flag ↦ ¬@.flag]]]
─────────────────────────────────────────────────────────────────────────
```

```
──────────────────────────── MODULE SpecLin ────────────────────────────
EXTENDS Library, RDR

CONSTANT Initial   TRUE when first instance.

VARIABLES
    status, pending, dState, initialized, abortVals, initVals, interface

INSTANCE SpecLinInterface

vars ≜ ⟨status, pending, dState, interface, initVals, initialized, abortVals⟩
statusStr ≜ {"idle", "ready", "aborted", "pending"}

TypeInvariant ≜
    ∧ ∀ p ∈ P :
        ∧ status[p] ∈ statusStr
        ∧ pending[p] ∈ Req
    ∧ dState ∈ S
    ∧ initVals ∈ SUBSET S
    ∧ abortVals ∈ SUBSET S

 Initial states
Init ≜
    ∧ IF Initial
        THEN   ∧ status = [p ∈ P ↦ "ready"]
               ∧ initialized  = TRUE
        ELSE   ∧ status = [p ∈ P ↦ "idle"]
               ∧ initialized  = FALSE
    ∧ dState = Bot
    ∧ pending = [p ∈ P ↦ Some(Req)]
    ∧ initVals = {}
    ∧ abortVals = {}
    ∧ interface  = InterfaceInit

 Invocation by process p:
Inv(p) ≜ ∃ c ∈ C :
     ∧ status[p] = "ready"
     ∧ status′ = [status EXCEPT ![p] = "pending"]
     ∧ pending′ = [pending EXCEPT ![p] = ⟨p, c⟩]
     ∧ Invoke(p, c)
     ∧ UNCHANGED ⟨dState, initialized, initVals, abortVals⟩

 Response by process p:
Resp(p) ≜
      ∧  status[p] = "pending"
      ∧  initialized
      ∧  status′ = [status EXCEPT ![p] = "ready"]
```

```
        $\land\ Contains(dState,\ pending[p])$
        $\land\ Response(p,\ Output(dState,\ pending[p]))$
        $\land\ \text{UNCHANGED}\ \langle dState,\ pending,\ initialized,\ initVals,\ abortVals \rangle$

$Pending\ \triangleq$
  $\{p \in P : status[p] \in \{\text{“pending”},\ \text{“aborted”}\}\}$
$PendingReqs\ \triangleq$
  $\{pending[p] : p \in Pending\}$
$InitSets\ \triangleq\ \text{SUBSET}\ initVals \setminus \{\{\}\}$

$SafeInit\ \triangleq$
  $\{s1 \in S :$
     $\land\ initVals \neq \{\}$
     $\land\ \exists\, is \in InitSets :$
       $\exists\, rs \in NoDupSeq(PendingReqs) :$
         $s1 = GLB(is) \star rs$
     $\land\ \forall\, a \in abortVals : s1 \preceq a\}$

$PossibleCommit\ \triangleq$
  $\{s1 \in S :$
     $\land\ dState \preceq s1$
     $\land\ \lor\ \exists\, rs \in NoDupSeq(PendingReqs) : s1 = dState \star rs$
        $\lor\ \exists\, is \in InitSets :$
          $\land\ dState \preceq GLB(is)$
          $\land\ \exists\, rs \in NoDupSeq(PendingReqs) : s1 = GLB(is) \star rs\}$

$SafeCommit\ \triangleq$
  $\{s1 \in PossibleCommit :$
    $\forall\, a \in abortVals : s1 \preceq a\}$

$SafeAbort\ \triangleq$
  $\{s1 \in S :$
    $\text{IF}\ initialized$
     $\text{THEN}\ s1 \in PossibleCommit$
     $\text{ELSE}\ \exists\, is \in InitSets :$
       $\exists\, rs \in NoDupSeq(PendingReqs) :$
         $s1 = GLB(is) \star rs\}$

 Abort by process $p$:
$Abo(p)\ \triangleq$
  $\land\ status[p] = \text{“pending”}$
  $\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“aborted”}]$
  $\land\ \exists\, s1 \in SafeAbort :$
    $\land\ Abort(p,\ pending[p][2],\ s1)$
    $\land\ abortVals' = abortVals \cup \{s1\}$
  $\land\ \text{UNCHANGED}\ \langle dState,\ pending,\ initialized,\ initVals \rangle$
```

```
Linearize some pending requests.
$Lin \triangleq$
    $\land\ initialized$
    $\land\ PendingReqs \neq \{\}$
    $\land\ \exists\, s \in SafeCommit : dState' = s$
    $\land\ dState' \in S$  For $TLC$
    $\land$ UNCHANGED $\langle status,\ pending,\ interface,\ initialized,\ initVals,\ abortVals \rangle$

$Init$ call
$Ini(p) \triangleq$
    $\land\ status[p] =$ "idle"
    $\land\ \exists\, c \in C,\ sval \in S :$
        $\land\ Initialize(p,\ c,\ sval)$
        $\land\ status' = [status$ EXCEPT $![p] =$ "pending"$]$
        $\land\ pending' = [pending$ EXCEPT $![p] = \langle p,\ c \rangle]$
        $\land\ initVals' = initVals \cup \{sval\}$
    $\land$ UNCHANGED $\langle dState,\ initialized,\ abortVals \rangle$

$Recover \triangleq$
    $\land\ \neg initialized$
    $\land\ \exists\, s1 \in SafeInit : dState' = s1$
    $\land\ dState' \in S$  For $TLC$
    $\land\ initialized' =$ TRUE
    $\land$ UNCHANGED $\langle pending,\ status,\ interface,\ initVals,\ abortVals \rangle$

$Next \triangleq \exists\, p \in P :\ Lin \lor Inv(p) \lor Resp(p) \lor Abo(p) \lor Ini(p) \lor Recover$

$Spec \triangleq Init \land \Box[Next]_{vars}$
```

```
┌──────────────── MODULE SpecLinCorrectness ────────────────┐
EXTENDS RDR

┌──────────────── MODULE SpecLinIsLin ────────────────┐
VARIABLE interface

Mode1(status, pending, dState, initialized, initVals, abortVals) ≜
    INSTANCE SpecLin WITH Initial ← TRUE

Lin(status, pending, dState, nxtOut) ≜ INSTANCE Linearizability WITH
     interface ← [inv ↦ interface.inv, resp ↦ interface.resp]

Mode1Spec ≜ ∃ status, pending, s, initVals, abortVals, initialized :
    Mode1(status, pending, s, initVals, abortVals, initialized)!Spec

LinSpec ≜ ∃ status, pending, s, nxtOut : Lin(status, pending, s, nxtOut)!Spec

THEOREM Mode1Spec ⇒ LinSpec
└─────────────────────────────────────────────────────┘

┌──────────────── MODULE SpecLinIsIdemPotent ────────────────┐
   Here we compose two instances of speculative linearizability using the method described in
   section 2.4.6.
EXTENDS SpecLinInterface

SingleMode(status, pending, dState, initVals, abortVals, initialized) ≜
    INSTANCE SpecLin WITH Initial ← TRUE

┌──────────────── MODULE Composition ────────────────┐
VARIABLES status1, pending1, dState1, initVals1, abortVals1, initialized1, interface1
vars1 ≜ ⟨status1, pending1, dState1, initVals1, abortVals1, initialized1,
  interface1⟩
Mode1 ≜ INSTANCE SpecLin WITH
    Initial ← TRUE,
    status ← status1, pending ← pending1, dState ← dState1, initVals ← initVals1,
    abortVals ← abortVals1,
    initialized ← initialized1, interface ← interface1

VARIABLES status2, pending2, dState2, initVals2, abortVals2, initialized2, interface2
vars2 ≜ ⟨status2, pending2, dState2, initVals2, abortVals2, initialized2,
  interface2⟩
Mode2 ≜ INSTANCE SpecLin WITH
    Initial ← FALSE,
    status ← status2, pending ← pending2, dState ← dState2, initVals ← initVals2,
    abortVals ← abortVals2,
```

$$\quad \mathit{initialized} \leftarrow \mathit{initialized}2,\ \mathit{interface} \leftarrow \mathit{interface}2$$

$$\begin{array}{l}
\mathit{LinkInterfaces} \triangleq \\
\quad \land\ \mathit{interface}1'.\mathit{abort} = \mathit{interface}2'.\mathit{init} \\
\quad \land\ \forall\, p \in P : \\
\qquad \land\ \mathit{interface}1'.\mathit{inv}[p] \neq \mathit{interface}1.\mathit{inv}[p] \\
\qquad\quad \Rightarrow \mathit{interface}.\mathit{inv}' = [\mathit{interface}.\mathit{inv}\ \text{EXCEPT}\ ![p] \\
\qquad\qquad\quad = [@\ \text{EXCEPT}\ !.\mathit{cmd} = \mathit{interface}1'.\mathit{inv}[p].\mathit{cmd}, \\
\qquad\qquad\qquad\qquad\qquad !.\mathit{flag} = \neg @]] \\
\qquad \land\ \mathit{interface}1'.\mathit{resp}[p] \neq \mathit{interface}1.\mathit{resp}[p] \\
\qquad\quad \Rightarrow \mathit{interface}.\mathit{resp}' = [\mathit{interface}.\mathit{resp}\ \text{EXCEPT}\ ![p] \\
\qquad\qquad\quad = [@\ \text{EXCEPT}\ !.\mathit{output} = \mathit{interface}1'.\mathit{resp}[p].\mathit{output}, \\
\qquad\qquad\qquad\qquad\qquad !.\mathit{flag} = \neg @]] \\
\qquad \land\ \mathit{interface}2'.\mathit{inv}[p] \neq \mathit{interface}2.\mathit{inv}[p] \\
\qquad\quad \Rightarrow \mathit{interface}.\mathit{inv}' = [\mathit{interface}.\mathit{inv}\ \text{EXCEPT}\ ![p] \\
\qquad\qquad\quad = [@\ \text{EXCEPT}\ !.\mathit{cmd} = \mathit{interface}2'.\mathit{inv}[p].\mathit{cmd}, \\
\qquad\qquad\qquad\qquad\qquad !.\mathit{flag} = \neg @]] \\
\qquad \land\ \mathit{interface}2'.\mathit{resp}[p] \neq \mathit{interface}2.\mathit{resp}[p] \\
\qquad\quad \Rightarrow \mathit{interface}.\mathit{resp}' = [\mathit{interface}.\mathit{resp}\ \text{EXCEPT}\ ![p] \\
\qquad\qquad\quad = [@\ \text{EXCEPT}\ !.\mathit{output} = \mathit{interface}2'.\mathit{resp}[p].\mathit{output}, \\
\qquad\qquad\qquad\qquad\qquad !.\mathit{flag} = \neg @]] \\
\quad \land\ \mathit{interface}1'.\mathit{interface}.\mathit{inv} = \mathit{interface}1.\mathit{interface}.\mathit{inv} \\
\quad \land\ \mathit{interface}2'.\mathit{interface}.\mathit{inv} = \mathit{interface}2.\mathit{interface}.\mathit{inv} \\
\qquad \Rightarrow \mathit{interface}.\mathit{inv}' = \mathit{interface}.\mathit{inv} \\
\quad \land\ \mathit{interface}1'.\mathit{interface}.\mathit{resp} = \mathit{interface}1.\mathit{interface}.\mathit{resp} \\
\quad \land\ \mathit{interface}2'.\mathit{interface}.\mathit{resp} = \mathit{interface}2.\mathit{interface}.\mathit{resp} \\
\qquad \Rightarrow \mathit{interface}.\mathit{resp}' = \mathit{interface}.\mathit{resp} \\
\quad \land\ \mathit{interface}1'.\mathit{abort} = \mathit{interface}2'.\mathit{init}
\end{array}$$

$$\begin{array}{l}
\mathit{CompoNext} \triangleq \\
\quad \land\ \lor\ \land\ \mathit{Mode}1!\mathit{Next} \\
\qquad\qquad \land\ \text{UNCHANGED}\ \mathit{vars}2 \\
\qquad \lor\ \land\ \mathit{Mode}2!\mathit{Next} \\
\qquad\qquad \land\ \text{UNCHANGED}\ \mathit{vars}1 \\
\qquad \lor\ \exists\, p \in P : \mathit{Mode}1!\mathit{Abo}(p) \land \mathit{Mode}2!\mathit{Ini}(p) \\
\quad \land\ \mathit{LinkInterfaces}
\end{array}$$

$$\mathit{CompoInit} \triangleq \mathit{Mode}1!\mathit{Init} \land \mathit{Mode}2!\mathit{Init} \land \mathit{interface} = \mathit{InterfaceInit}$$

$$\mathit{CompoSpec} \triangleq \mathit{CompoInit} \land \Box[\mathit{CompoNext}]_{\langle \mathit{vars}1,\, \mathit{vars}2,\, \mathit{interface} \rangle}$$

$$\begin{array}{l}
\mathit{Compo}(\mathit{status}1,\ \mathit{pending}1,\ \mathit{dState}1,\ \mathit{initVals}1,\ \mathit{abortVals}1,\ \mathit{initialized}1,\ \mathit{interface}1, \\
\quad \mathit{status}2,\ \mathit{pending}2,\ \mathit{dState}2,\ \mathit{initVals}2,\ \mathit{abortVals}2,\ \mathit{initialized}2,\ \mathit{interface}2) \\
\qquad\quad \triangleq\ \text{INSTANCE}\ \mathit{Composition}
\end{array}$$

$$
\begin{array}{l}
\mathit{CompoSpec} \triangleq \\
\quad \exists\, \mathit{status}1,\ \mathit{pending}1,\ \mathit{dState}1,\ \mathit{initVals}1,\ \mathit{abortVals}1,\ \mathit{initialized}1,\ \mathit{interface}1 : \\
\quad\quad \exists\, \mathit{status}2,\ \mathit{pending}2,\ \mathit{dState}2,\ \mathit{initVals}2,\ \mathit{abortVals}2,\ \mathit{initialized}2,\ \mathit{interface}2 : \\
\quad\quad\quad \mathit{Compo}(\mathit{status}1,\ \mathit{pending}1,\ \mathit{dState}1,\ \mathit{initVals}1,\ \mathit{abortVals}1,\ \mathit{initialized}1, \\
\quad\quad\quad\quad \mathit{interface}1,\ \mathit{status}2,\ \mathit{pending}2,\ \mathit{dState}2,\ \mathit{initVals}2,\ \mathit{abortVals}2, \\
\quad\quad\quad\quad\quad \mathit{initialized}2,\ \mathit{interface}2)\,!\,\mathit{CompoSpec}
\end{array}
$$

$$
\begin{array}{l}
\mathit{SingleModeSpec} \triangleq \\
\quad \exists\, \mathit{status},\ \mathit{pending},\ \mathit{dState},\ \mathit{initVals},\ \mathit{abortVals},\ \mathit{initialized} : \\
\quad\quad \mathit{SingleMode}(\mathit{status},\ \mathit{pending},\ \mathit{dState},\ \mathit{initVals},\ \mathit{abortVals},\ \mathit{initialized})\,!\,\mathit{Spec}
\end{array}
$$

THEOREM $\mathit{CompoSpec} \Rightarrow \mathit{SingleModeSpec}$

## A.2 Message-Passing Adaptive Algorithms

```
┌──────────────────── MODULE MPGC ────────────────────┐
   A specification that factorizes the common structure of fast and safe modes. The acceptors
   execute arbitrary proposed requests that they read directly from the array pending. The
   acceptors can also not accept any request and set their dState to None. Clients consider their
   request executed when it appears in the GLB of a RespQuorum of acceptors. The clients abort
   with an abort value computed from the dstates of an AbortQuorum of acceptors using the
   operator AbortValues(_).
 EXTENDS RDR, Library

 CONSTANTS Initial, Acceptor

 CONSTANTS RespQuorum, AbortQuorum, AbortValues(_), None
 ASSUME None ∉ S


 VARIABLES status, pending, initVals, dState, accStatus, interface,
   pastPending
  abortVals is a history variable
 VARIABLE abortVals

 INSTANCE SpecLinInterface

 vars ≜ ⟨status, pending, initVals, dState, accStatus, interface,
   abortVals, pastPending⟩
 Labels ≜ {"idle", "ready", "pending", "panic", "aborted"}
 AcceptorLabels ≜ {"idle", "ready", "stopped"}

 TypeInvariant ≜
   ∧ ∀ p ∈ P :
      ∧ status[p] ∈ Labels
      ∧ pending[p] ∈ Req
   ∧ ∀ r ∈ Acceptor :
      ∧ dState[r] ∈ S ∪ {None}
      ∧ accStatus[r] ∈ AcceptorLabels
   ∧ initVals ⊆ S
   ∧ pastPending ⊆ Req

 Init ≜
   ∧ status =
     IF Initial
       THEN [p ∈ P ↦ "ready"]
       ELSE  [p ∈ P ↦ "idle"]
   ∧ pending = [p ∈ P ↦ Some(Req)]
   ∧ initVals = {}
   ∧ dState = [r ∈ Acceptor ↦ Bot]
   ∧ accStatus =
     IF Initial
       THEN [r ∈ Acceptor ↦ "ready"]
      $\text{ELSE} \ [r \in Acceptor \mapsto \text{“idle”}]$
  $\land\ interface\ = InterfaceInit$
  $\land\ abortVals = \{\}$
  $\land\ pastPending = \{\}$

$Ini(p)\ \triangleq\ \exists\, c \in C,\ v \in S :$
  $\land\ status[p]\ = \text{“idle”}$
  $\land\ pending' = [pending\ \text{EXCEPT}\ ![p] = \langle p,\ c \rangle]$
  $\land\ initVals' =\ \ initVals \cup \{v\}$
  $\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“pending”}]$
  $\land\ Initialize(p,\ c,\ v)$
  $\land\ pastPending' = pastPending \cup \{\langle p,\ c \rangle\}$
  $\land\ \text{UNCHANGED}\ \langle dState,\ accStatus,\ abortVals \rangle$

$Inv(p)\ \triangleq\ \exists\, c \in C :$
  $\land\ status[p]\ = \text{“ready”}$
  $\land\ pending' = [pending\ \text{EXCEPT}\ ![p] = \langle p,\ c \rangle]$
  $\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“pending”}]$
  $\land\ Invoke(p,\ c)$
  $\land\ pastPending' = pastPending \cup \{\langle p,\ c \rangle\}$
  $\land\ \text{UNCHANGED}\ \langle dState,\ accStatus,\ initVals,\ abortVals \rangle$

$AccStates(Q)\ \triangleq$
  $\{s \in S : \exists\, srv \in Q : s = dState[srv]\}$

$ValidRespQuorum\ \triangleq$
  $\{Q \in RespQuorum : \forall\, a \in Q : dState[a] \neq None\}$

$Res(p)\ \triangleq$
  $\land\ status[p] = \text{“pending”}$
  $\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“ready”}]$
  $\land\ \exists\, Q \in ValidRespQuorum :$
    $\land\ \forall\, srv \in Q :$
      $\land\ accStatus[srv] \neq \text{“idle”}$
    $\land\ \text{LET}\ glb\ \triangleq\ GLB(AccStates(Q))$
       $\text{IN}\quad \land\ Contains(glb,\ pending[p])$
             $\land\ Response(p,\ Output(glb,\ pending[p]))$
  $\land\ \text{UNCHANGED}\ \langle pending,\ initVals,\ dState,\ accStatus,$
    $abortVals,\ pastPending \rangle$

$ValidAbortQuorum\ \triangleq$
  $\{Q \in AbortQuorum : \forall\, a \in Q : dState[a] \neq None\}$

$Abo(p)\ \triangleq$
  $\land\ status[p] = \text{“panic”}$
  $\land$
    $\lor\ \exists\, Q \in ValidAbortQuorum :$
```

```
            $\land\ \forall\, srv \in Q :$
                $\land\ accStatus[srv] = \text{“stopped”}$
            $\land\ \exists\, s \in AbortValues([a \in Q \mapsto dState[a]]) :$
                $\land\ Abort(p, pending[p][2], s)$
                $\land\ abortVals' = abortVals \cup \{s\}$
        $\lor\ \exists\, Q \in AbortQuorum :$
            $\land\ \forall\, srv \in Q :$
                $\land\ accStatus[srv] = \text{“stopped”}$
                $\land\ dState[srv] = None$
            $\land\ \exists\, s \in initVals :$
                $\land\ Abort(p, pending[p][2], s)$
                $\land\ abortVals' = abortVals \cup \{s\}$
    $\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“aborted”}]$
    $\land\ \text{UNCHANGED}\ \langle pending, initVals, dState, accStatus, pastPending \rangle$

We abstract over time: a process can panic at any moment.
$Panic(p) \triangleq$
    $\land\ status[p] = \text{“pending”}$
    $\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“panic”}]$
    $\land\ \text{UNCHANGED}\ \langle pending, initVals, dState, accStatus, interface,$
      $abortVals, pastPending \rangle$

$Pending \triangleq$
    $\{p \in P : status[p] \in \{\text{“pending”}, \text{“panic”}, \text{“aborted”}\}\}$

An $Acceptor$ executes a pending request.
$Exec(r) \triangleq$
    $\land\ accStatus[r] = \text{“ready”}$
    $\land\ \exists\, req \in pastPending :$
        $\land\ dState' = [dState\ \text{EXCEPT}\ ![r] = @ \bullet req]$
        $\land\ dState'[r] \in S$   For $TLC$
    $\land\ \text{UNCHANGED}\ \langle status, pending, initVals, accStatus, interface,$
      $abortVals, pastPending \rangle$

An $Acceptor$ sets its local state to one of the init values of the processes.
$WakeUp(r) \triangleq$
    $\land\ accStatus[r] = \text{“idle”}$
    $\land\ \exists\, iv \in initVals :$
        $\land\ dState' = [dState\ \text{EXCEPT}\ ![r] = iv]$
        $\land\ accStatus' = [accStatus\ \text{EXCEPT}\ ![r] = \text{“ready”}]$
    $\land\ \text{UNCHANGED}\ \langle status, pending, initVals, interface, abortVals, pastPending \rangle$

$Timeout(r) \triangleq$
    $\land\ accStatus[r] = \text{“idle”}$
    $\land\ dState' = [dState\ \text{EXCEPT}\ ![r] = None]$
    $\land\ accStatus' = [accStatus\ \text{EXCEPT}\ ![r] = \text{“stopped”}]$
```

$$\land \textsc{unchanged}\ \langle status, pending, initVals, interface, abortVals, pastPending\rangle$$

A replica stops when at least one client has panicked

$$
\begin{aligned}
&Stop(r) \triangleq \exists\, p \in P : \\
&\qquad \land\ status[p] \in \{\text{“panic”}, \text{“aborted”}\} \\
&\qquad \land\ accStatus[r] = \text{“ready”} \\
&\qquad \land\ accStatus' = [accStatus\ \textsc{except}\ ![r] = \text{“stopped”}] \\
&\qquad \land\ \textsc{unchanged}\ \langle status, pending, initVals, interface, dState, \\
&\qquad\quad abortVals, pastPending\rangle
\end{aligned}
$$

$$
\begin{aligned}
&Next \triangleq \\
&\qquad \lor\ \exists\, p \in P : Ini(p) \lor Inv(p) \lor Res(p) \lor Abo(p) \lor Panic(p) \\
&\qquad \lor\ \exists\, r \in Acceptor : Exec(r) \lor WakeUp(r) \lor Stop(r) \lor Timeout(r)
\end{aligned}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

```
──────────────── MODULE FastMPGCDefs ────────────────
EXTENDS FiniteSets, Library

CONSTANTS _ ⪯ _, GLB(_)

CONSTANTS RespQuorum, AbortQuorum, Acceptor


ASSUME ∀ Q, R ∈ RespQuorum : Q ∩ R ≠ {}
ASSUME ∀ Q ∈ RespQuorum, R ∈ AbortQuorum :
  Cardinality(Q ∩ R) ≥ (Cardinality(R) ÷ 2) + 1

 Examples of quorums:
N ≜ Cardinality(Acceptor)
RespQuorum1 ≜
  {Q ∈ SUBSET Acceptor :
    Cardinality(Q) ≥ ((2 ∗ N) ÷ 3) + 1}
AbortQuorum1 ≜ RespQuorum1
RespQuorum2 ≜
  {Q ∈ SUBSET Acceptor :
    Cardinality(Q) ≥ ((3 ∗ N) ÷ 4) + 1}
AbortQuorum2 ≜
  {Q ∈ SUBSET Acceptor :
    Cardinality(Q) ≥ (N ÷ 2) + 1}
RespQuorum3 ≜ {Acceptor}
AbortQuorum3 ≜ {{a} : a ∈ Acceptor}


RemovePrefixes(ss) ≜
  {s ∈ ss : ¬(∃ s1 ∈ ss \ {s} : s ⪯ s1)}

Majority(Q) ≜
  {Maj ∈ SUBSET Q :
    Cardinality(Maj) ≥ Cardinality(Q) ÷ 2 + 1}

 Assumes DStates is a function [Q → S] where S is an abort quorum.
AbortValues(DStates) ≜
  LET Q ≜ DOMAIN DStates
      MajSets ≜ {Image([a ∈ Maj ↦ DStates[a]]) : Maj ∈ Majority(Q)}
       G ≜ {GLB(ss) : ss ∈ MajSets}
  IN   RemovePrefixes(G)
─────────────────────────────────────────────────────
```

```
──────────────── MODULE FastMPGC ────────────────
 The specification of fast modes

EXTENDS FiniteSets, Naturals, Library, Generic, TLCDefs

CONSTANTS Initial, Acceptor

CONSTANTS RespQuorum, AbortQuorum, None

INSTANCE FastMPGCDefs

VARIABLES status, pending, initVals, dState, accStatus, interface,
  abortVals, pastPending

INSTANCE MPGC

slin_status ≜ [p ∈ P ↦ IF status[p] ∈ {"pending", "panic"} THEN "pending" ELSE status[p]]
slin_pending ≜ pending
slin_dState ≜ Max({GLB(AccStates(Q)) : Q ∈ RespQuorum}, LAMBDA a, b : a ⪯ b)
slin_interface ≜ interface
slin_initialized ≜
  IF Initial THEN TRUE
   ELSE ∃ Q ∈ RespQuorum : ∀ a ∈ Q : accStatus[a] ≠ "idle"
slin_initVals ≜ initVals
slin_abortVals ≜ abortVals

SLin ≜ INSTANCE SpecLin WITH
    status ← slin_status,
    pending ← slin_pending,
    dState ← slin_dState,
    interface ← slin_interface,
    initialized ← slin_initialized,
    initVals ← slin_initVals,
    abortVals ← slin_abortVals

THEOREM Spec ⇒ SLin!Spec
─────────────────────────────────────────────────
```

```
──────────────── MODULE SafeMPGCDefs ────────────────
EXTENDS FiniteSets, Library

CONSTANTS _ ⪯ _, GLB(_)

CONSTANTS Acceptor

N ≜ Cardinality(Acceptor)
Quorum ≜
  {Q ∈ SUBSET Acceptor :
     Cardinality(Q) ≥ (N ÷ 2) + 1}
RespQuorum ≜ Quorum
AbortQuorum ≜ Quorum

ASSUME ∀ Q, R ∈ RespQuorum : Q ∩ R ≠ {}
ASSUME ∀ Q ∈ RespQuorum, R ∈ AbortQuorum :
  Cardinality(Q ∩ R) ≥ (Cardinality(R) ÷ 2) + 1

 There is only one abort value, the maximum of the dStates of the quorum.
AbortValues(DStates) ≜
  {CHOOSE s ∈ Image(DStates) :
     ∀ s1 ∈ Image(DStates) :
       s1 ⪯ s}
─────────────────────────────────────────────────────
```

```
──────────────────── MODULE SafeMPGC ────────────────────
  The specification of safe modes

EXTENDS FiniteSets, Naturals, Library, Consensus, TLCDefs

CONSTANTS Initial, Acceptor, None

INSTANCE SafeMPGCDefs

VARIABLES status, pending, initVals, dState, accStatus, interface,
  abortVals, pastPending

  In safe algorithms, the acceptors cannot become inconsistent. This can
  be implemented with a leader, or otherwise.
AcceptorConsistency ≜
  ∀ acc1, acc2 ∈ Acceptor :
    LET s1 ≜ dState[acc1]
        s2 ≜ dState[acc2]
    IN
      s1 ≠ None ∧ s2 ≠ None
       ⇒ s1 ⪯ s2 ∨ s2 ⪯ s1


INSTANCE MPGC

ConsistentSpec ≜ Init ∧ □[Next ∧ AcceptorConsistency′]_vars

slin_status ≜
  [p ∈ P ↦
    IF status[p] ∈ {"pending", "panic"}
     THEN "pending"
     ELSE status[p]]
slin_pending ≜
  pending
slin_dState ≜
  IF ValidRespQuorum ≠ {}
   THEN
    Max(
    {GLB(AccStates(Q)) :
      Q ∈ ValidRespQuorum},
    LAMBDA a, b : a ⪯ b)
   ELSE Bot
slin_interface ≜
  interface
slin_initialized ≜
  IF Initial THEN TRUE
   ELSE ∃ Q ∈ RespQuorum :
    ∀ a ∈ Q :
        ∧ accStatus[a] ≠ "idle"
        ∧ dState[a] ≠ None
slin_initVals ≜
  initVals
slin_abortVals ≜
  abortVals

SLin ≜ INSTANCE SpecLin WITH
    status ← slin_status,
    pending ← slin_pending,
    dState ← slin_dState,
    interface ← slin_interface,
    initialized ← slin_initialized,
    initVals ← slin_initVals,
    abortVals ← slin_abortVals

THEOREM ConsistentSpec ⇒ SLin!Spec
```

```
──────────────────────── MODULE Network ────────────────────────
CONSTANT Msg, Agent

Packet ≜ [from : Agent, to : Agent, msg : Msg]
MkPacket(src, m, dst) ≜ [
    from ↦ src,
    msg ↦ m,
    to ↦ dst]

VARIABLE network

  Async channels with message loss
  but no duplication or corruption (e.g. TCP).

Packets(src, ms) ≜
    UNION {{MkPacket(src, m, dst) : m ∈ ms[dst]} : dst ∈ DOMAIN ms}
  ms must be a function from destination to set of messages.
Snd(src, ms) ≜
     network′ = network ∪ Packets(src, ms)

Rcv(dst, m, src) ≜
     LET packet ≜ MkPacket(src, m, dst)
     IN   ∧ packet ∈ network
          ∧ network′ = network \ {packet}

  Receive and reply at once, to simplify.
RcvSnd(dst, m, src, ms) ≜
    LET packet ≜ MkPacket(src, m, dst)
    IN   ∧ packet ∈ network
         ∧ network′ = (network \ {packet}) ∪ Packets(dst, ms)

NetworkInvariant ≜
  ∀ packet ∈ network :
    packet ∈ Packet
────────────────────────────────────────────────────────────────
```

```
──────────────────────── MODULE Quorum ────────────────────────
 The Quorum Algorithm

EXTENDS Consensus, Library, TLCDefs
INSTANCE RDR

CONSTANTS AbortQuorum, RespQuorum, Initial, Acceptor, None

INSTANCE FastMPGCDefs

VARIABLES status, pending, initVals, execAcks, panicAcks, dState,
  network, accStatus, interface
 abortVals and pastPending are history variables
VARIABLE abortVals, pastPending

INSTANCE SpecLinInterface

vars ≜ ⟨status, pending, initVals, execAcks, panicAcks, dState,
  accStatus, interface, network, abortVals, pastPending⟩

Labels ≜ {"idle", "ready", "pending", "panic", "aborted"}
AcceptorLabels ≜ {"idle", "ready", "stopped"}

Agent ≜ P ∪ Acceptor

Msg ≜ {⟨"req", r⟩ : r ∈ Req} ∪ {⟨"execAck", s⟩ : s ∈ S}
   ∪ {⟨"panic"⟩} ∪ {⟨"panicAck", s⟩ : s ∈ S}
   ∪ {⟨"init", s⟩ : s ∈ S}
INSTANCE Network

TypeInvariant ≜
   ∧ ∀ p ∈ P :
      ∧ status[p] ∈ Labels
      ∧ pending[p] ∈ Req
      ∧ ∀ a ∈ Acceptor :
         ∧ execAcks[p][a] ∈ {{s} : s ∈ S} ∪ {{}}
         ∧ panicAcks[p][a] ∈ {{s} : s ∈ S} ∪ {{}}
   ∧ ∀ a ∈ Acceptor :
      ∧  dState[a] ∈ S ∪ {None}
      ∧  accStatus[a] ∈ AcceptorLabels
   ∧ initVals ⊆ S
   ∧ abortVals ⊆ S
   ∧ pastPending ⊆ Req

 The processes

InitProcs ≜
   ∧  status = [p ∈ P ↦ IF Initial THEN "ready" ELSE  "idle"]
```

```
    $\land\ \mathit{pending} = [p \in P \mapsto \mathit{Some}(\mathit{Req})]$
    $\land\ \mathit{execAcks} = [p \in P \mapsto [a \in \mathit{Acceptor} \mapsto \{\}]]$
    $\land\ \mathit{panicAcks} = [p \in P \mapsto [a \in \mathit{Acceptor} \mapsto \{\}]]$

$\mathit{Inv}(p) \triangleq$
    $\land\ \mathit{status}[p] = \text{“ready”}$
    $\land\ \exists\, c \in C :$
        $\land\ \mathit{Invoke}(p,\ c)$
        $\land\ \mathit{Snd}(p,\ [a \in \mathit{Acceptor} \mapsto \{\langle \text{“req”},\ \langle p,\ c \rangle\rangle\}])$
        $\land\ \mathit{pending}' = [\mathit{pending}\ \text{EXCEPT}\ ![p] = \langle p,\ c \rangle]$
        $\land\ \mathit{pastPending}' = \mathit{pastPending} \cup \{\langle p,\ c \rangle\}$
    $\land\ \mathit{status}' = [\mathit{status}\ \text{EXCEPT}\ ![p] = \text{“pending”}]$
    $\land\ \text{UNCHANGED}\ \langle \mathit{initVals},\ \mathit{execAcks},\ \mathit{panicAcks},\ \mathit{dState},\ \mathit{accStatus},$
      $\mathit{abortVals} \rangle$

$\mathit{Ini}(p) \triangleq$
    $\land\ \mathit{status}[p] = \text{“idle”}$
    $\land\ \exists\, c \in C,\ s \in S :$
        $\land\ \mathit{Initialize}(p,\ c,\ s)$
        $\land\ \mathit{Snd}(p,\ [a \in \mathit{Acceptor} \mapsto \{\langle \text{“init”},\ s \rangle,\ \langle \text{“req”},\ \langle p,\ c \rangle\rangle\}])$
        $\land\ \mathit{pending}' = [\mathit{pending}\ \text{EXCEPT}\ ![p] = \langle p,\ c \rangle]$
        $\land\ \mathit{initVals}' = \mathit{initVals} \cup \{s\}$
        $\land\ \mathit{pastPending}' = \mathit{pastPending} \cup \{\langle p,\ c \rangle\}$
    $\land\ \mathit{status}' = [\mathit{status}\ \text{EXCEPT}\ ![p] = \text{“pending”}]$
    $\land\ \text{UNCHANGED}\ \langle \mathit{execAcks},\ \mathit{panicAcks},\ \mathit{dState},\ \mathit{accStatus},\ \mathit{abortVals} \rangle$

$\mathit{RcvExecAcC}(p) \triangleq$
    $\land\ \mathit{status}[p] \in \{\text{“pending”},\ \text{“panic”}\}$
    $\land\ \exists\, s \in S,\ a \in \mathit{Acceptor} :$
        $\land\ \mathit{Rcv}(p,\ \langle \text{“execAck”},\ s \rangle,\ a)$
        $\land\ \mathit{execAcks}' = [\mathit{execAcks}\ \text{EXCEPT}\ ![p] = [@\ \text{EXCEPT}\ ![a] = \{s\}]]$
    $\land\ \text{UNCHANGED}\ \langle \mathit{status},\ \mathit{pending},\ \mathit{initVals},\ \mathit{dState},\ \mathit{accStatus},$
      $\mathit{interface},\ \mathit{abortVals},\ \mathit{panicAcks},\ \mathit{pastPending} \rangle$

$\mathit{RcvPanicAck}(p) \triangleq$
    $\land\ \mathit{status}[p] = \text{“panic”}$
    $\land\ \exists\, s \in S,\ a \in \mathit{Acceptor} :$
        $\land\ \mathit{Rcv}(p,\ \langle \text{“panicAck”},\ s \rangle,\ a)$
        $\land\ \mathit{panicAcks}' = [\mathit{panicAcks}\ \text{EXCEPT}\ ![p] = [@\ \text{EXCEPT}\ ![a] = \{s\}]]$
    $\land\ \text{UNCHANGED}\ \langle \mathit{status},\ \mathit{pending},\ \mathit{execAcks},\ \mathit{initVals},\ \mathit{dState},$
      $\mathit{accStatus},\ \mathit{interface},\ \mathit{abortVals},\ \mathit{pastPending} \rangle$

$\mathit{Panic}(p) \triangleq$
    $\land\ \mathit{status}[p] = \text{“pending”}$
    $\land\ \mathit{status}' = [\mathit{status}\ \text{EXCEPT}\ ![p] = \text{“panic”}]$
    $\land\ \mathit{Snd}(p,\ [a \in \mathit{Acceptor} \mapsto \{\langle \text{“panic”} \rangle\}])$
```

```
  ∧  UNCHANGED ⟨pending, initVals, execAcks, panicAcks, dState,
    accStatus, interface, abortVals, pastPending⟩

Res(p) ≜
  ∧ status[p] = "pending"
  ∧ ∃ Q ∈ RespQuorum :
    ∧ ∀ a ∈ Q : execAcks[p][a]  ≠ {}
    ∧ LET acks ≜ {s ∈ S : ∃ a ∈ Q : execAcks[p][a] = {s}}
          glb  ≜ GLB(acks)
           req ≜ pending[p]
      IN    ∧ Contains(glb, req)
            ∧ Response(p, Output(glb, req))
  ∧ status′ = [status EXCEPT ![p] = "ready"]
  ∧ execAcks′ = [execAcks EXCEPT ![p] = [a ∈ Acceptor ↦ {}]]
  ∧ UNCHANGED ⟨pending, initVals, panicAcks, dState, accStatus,
    network, abortVals, pastPending⟩

PanicAck(p, a) ≜
  CHOOSE s ∈ S : panicAcks[p][a] = {s}

Abo(p) ≜
  ∧ status[p] = "panic"
  ∧ ∃ Q ∈ AbortQuorum :
    ∧ ∀ a ∈ Q : panicAcks[p][a] ≠ {}
    ∧ LET acks ≜ [a ∈ Q ↦ PanicAck(p, a)]
      IN  ∃ s ∈ AbortValues(acks) :
            ∧ Abort(p, pending[p][2], s)
            ∧ abortVals′ = abortVals ∪ {s}
  ∧ status′ = [status EXCEPT ![p] = "aborted"]
  ∧ UNCHANGED ⟨pending, initVals, execAcks, panicAcks, dState, accStatus,
    network, pastPending⟩

 The Acceptors

InitAcceptor ≜
  ∧ accStatus = [a ∈ Acceptor ↦ IF Initial THEN "ready" ELSE  "idle"]
  ∧ dState = [a ∈ Acceptor ↦ Bot]

WakeUp(a) ≜
  ∧ accStatus[a] = "idle"
  ∧ accStatus′ = [accStatus EXCEPT ![a] = "ready"]
  ∧ ∃ p ∈ P, s ∈ S :
    ∧ Rcv(a, ⟨"init", s⟩, p)
    ∧ dState′ = [dState EXCEPT ![a] = s]
  ∧ UNCHANGED ⟨status, initVals, panicAcks, pending, execAcks,
    interface, abortVals, pastPending⟩
```

```
$Exec(a) \triangleq$
  $\land\ accStatus[a] = \text{“ready”}$
  $\land\ \exists\, p \in P,\ req \in Req :$
    $\land\ RcvSnd(a, \langle \text{“req”}, req\rangle, p,$
            $[q\ \in \{p\} \mapsto \{\langle \text{“execAck”}, dState[a] \bullet req\rangle\}])$
    $\land\ dState' = [dState\ \text{EXCEPT}\ ![a] = @ \bullet req]$
    $\land\ dState'[a] \in S$  For $TLC$
  $\land\ \text{UNCHANGED}\ \langle status, initVals, pending, execAcks, interface,$
   $accStatus, abortVals, panicAcks, pastPending\rangle$

$Stop(a) \triangleq$
  $\land\ accStatus[a] = \text{“ready”}$
  $\land\ \exists\, p \in P : RcvSnd(a, \langle \text{“panic”}\rangle, p,$
   $[q \in \{p\} \mapsto \{\langle \text{“panicAck”}, dState[a]\rangle\}])$
  $\land\ accStatus' = [accStatus\ \text{EXCEPT}\ ![a] = \text{“stopped”}]$
  $\land\ \text{UNCHANGED}\ \langle status, initVals, pending, execAcks, interface, dState,$
   $abortVals, panicAcks, pastPending\rangle$

 The full spec

$Init \triangleq$
  $\land\ InitProcs$
  $\land\ InitAcceptor$
  $\land\ interface = InterfaceInit$
  $\land\ network = \{\}$
  $\land\ abortVals = \{\}$
  $\land\ initVals = \{\}$
  $\land\ pastPending = \{\}$

$Next \triangleq$
  $\lor\ \exists\, p \in P : Inv(p) \lor Ini(p) \lor RcvPanicAck(p) \lor RcvExecAcC(p)$
    $\lor\ Panic(p) \lor Abo(p) \lor Res(p)$
  $\lor\ \exists\, a \in Acceptor : WakeUp(a) \lor Exec(a) \lor Stop(a)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Fast \triangleq \text{INSTANCE}\ FastMPGC$

$\text{THEOREM}\ Spec \Rightarrow Fast!Spec$
```

## MODULE *ZLight*

```
EXTENDS $Consensus, Library, TLCDefs$
INSTANCE $RDR$

CONSTANTS $Initial, Leader, Follower, None$
ASSUME $Leader \notin Follower$

$Acceptor \triangleq Follower \cup \{Leader\}$

INSTANCE $SafeMPGCDefs$

VARIABLES $status, pending, initVals, execAcks, panicAcks, dState,$
  $network, accStatus, interface$
 $abortVals$ and $pastPending$ are history variables
VARIABLE $abortVals, pastPending$

INSTANCE $SpecLinInterface$

$vars \triangleq \langle status, pending, initVals, execAcks, panicAcks, dState,$
  $accStatus, interface, network, abortVals, pastPending \rangle$

$Labels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“pending”}, \text{“panic”}, \text{“aborted”}\}$
$AcceptorLabels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“stopped”}\}$

$Agent \triangleq P \cup Acceptor$

$Msg \triangleq \{\langle \text{“req”}, r \rangle : r \in Req\} \cup \{\langle \text{“execAck”}, s \rangle : s \in S\}$
      $\cup \{\langle \text{“panic”} \rangle\} \cup \{\langle \text{“panicAck”}, s \rangle : s \in S\}$
      $\cup \{\langle \text{“init”}, s \rangle : s \in S\}$
      $\cup \{\langle \text{“leaderInit”}, s \rangle : s \in S\}$
      $\cup \{\langle \text{“leaderExec”}, s, p \rangle : s \in S, p \in P\}$
INSTANCE $Network$

$TypeInvariant \triangleq$
  $\land \forall p \in P :$
     $\land status[p] \in Labels$
     $\land pending[p] \in Req$
     $\land \forall a \in Acceptor :$
        $\land execAcks[p][a] \in \{\{s\} : s \in S\} \cup \{\{\}\}$
        $\land panicAcks[p][a] \in \{\{s\} : s \in S\} \cup \{\{\}\}$
  $\land \forall a \in Acceptor :$
     $\land\ dState[a] \in S \cup \{None\}$
     $\land\ accStatus[a] \in AcceptorLabels$
  $\land initVals \subseteq S$
  $\land abortVals \subseteq S$
  $\land pastPending \subseteq Req$

 The processes
```

```
$InitProcs \triangleq$
  $\land\ status = [p \in P \mapsto \text{IF } Initial \text{ THEN } \text{“ready”} \text{ ELSE } \text{“idle”}]$
  $\land\ pending = [p \in P \mapsto Some(Req)]$
  $\land\ execAcks = [p \in P \mapsto [a \in Acceptor \mapsto \{\}]]$
  $\land\ panicAcks = [p \in P \mapsto [a \in Acceptor \mapsto \{\}]]$

$Inv(p) \triangleq$
    $\land\ status[p] = \text{“ready”}$
    $\land\ \exists\, c \in C :$
        $\land\ Invoke(p, c)$
        $\land\ Snd(p, [a \in \{Leader\} \mapsto \{\langle \text{“req”}, \langle p, c \rangle \rangle\}])$
        $\land\ pending' = [pending \text{ EXCEPT } ![p] = \langle p, c \rangle]$
        $\land\ pastPending' = pastPending \cup \{\langle p, c \rangle\}$
    $\land\ status' = [status \text{ EXCEPT } ![p] = \text{“pending”}]$
    $\land\ \text{UNCHANGED } \langle initVals, abortVals, execAcks, dState, accStatus, panicAcks \rangle$

$Ini(p) \triangleq$
    $\land\ status[p] = \text{“idle”}$
    $\land\ \exists\, c \in C, s \in S :$
        $\land\ Initialize(p, c, s)$
        $\land\ Snd(p, [a \in \{Leader\} \mapsto \{\langle \text{“init”}, s \rangle, \langle \text{“req”}, \langle p, c \rangle \rangle\}])$
        $\land\ pending' = [pending \text{ EXCEPT } ![p] = \langle p, c \rangle]$
        $\land\ initVals' = initVals \cup \{s\}$
        $\land\ pastPending' = pastPending \cup \{\langle p, c \rangle\}$
    $\land\ status' = [status \text{ EXCEPT } ![p] = \text{“pending”}]$
    $\land\ \text{UNCHANGED } \langle execAcks, dState, accStatus, abortVals, panicAcks \rangle$

$RcvExecAcC(p) \triangleq$
  $\land\ status[p] \in \{\text{“pending”}, \text{“panic”}\}$
  $\land\ \exists\, s \in S, a \in Acceptor :$
    $\land\ Rcv(p, \langle \text{“execAck”}, s \rangle, a)$
    $\land\ execAcks' = [execAcks \text{ EXCEPT } ![p] = [@ \text{ EXCEPT } ![a] = \{s\}]]$
  $\land\ \text{UNCHANGED } \langle status, pending, initVals, dState, accStatus,$
    $interface, abortVals, panicAcks, pastPending \rangle$

$RcvPanicAck(p) \triangleq$
  $\land\ status[p] = \text{“panic”}$
  $\land\ \exists\, s \in S, a \in Acceptor :$
    $\land\ Rcv(p, \langle \text{“panicAck”}, s \rangle, a)$
    $\land\ panicAcks' = [panicAcks \text{ EXCEPT } ![p] = [@ \text{ EXCEPT } ![a] = \{s\}]]$
  $\land\ \text{UNCHANGED } \langle status, pending, execAcks, initVals, dState,$
    $accStatus, interface, abortVals, pastPending \rangle$

 A process can panic at any time because it times out.
$Panic(p) \triangleq$
  $\land\ status[p] = \text{“pending”}$
  ∧  status′ = [status EXCEPT ![p] = “panic”]
  ∧  Snd(p, [a ∈ Acceptor ↦ {⟨“panic”⟩}])
  ∧  UNCHANGED ⟨pending, initVals, execAcks, panicAcks, dState,
     accStatus, interface, abortVals, pastPending⟩

Res(p) ≜
  ∧ status[p] = “pending”
  ∧ ∃ Q ∈ RespQuorum :
     ∧ ∀ a ∈ Q : execAcks[p][a]  ≠ {}
     ∧ LET acks ≜ {s ∈ S : ∃ a ∈ Q : execAcks[p][a] = {s}}
           glb  ≜ GLB(acks)
            req ≜ pending[p]
        IN    ∧  Contains(glb, req)
              ∧  Response(p, Output(glb, req))
  ∧ status′ = [status EXCEPT ![p] = “ready”]
  ∧ execAcks′ = [execAcks EXCEPT ![p] = [a ∈ Acceptor ↦ {}]]
  ∧ UNCHANGED ⟨pending, initVals, panicAcks, dState, accStatus,
    network, abortVals, pastPending⟩

PanicAck(p, a) ≜
  CHOOSE s ∈ S : panicAcks[p][a] = {s}

Abo(p) ≜
  ∧ status[p] = “panic”
  ∧ ∃ Q ∈ AbortQuorum :
     ∧ ∀ a ∈ Q : panicAcks[p][a] ≠ {}
     ∧ LET acks ≜ [a ∈ Q ↦ PanicAck(p, a)]
        IN   ∃ s ∈ AbortValues(acks) :
                 ∧ Abort(p, pending[p][2], s)
                 ∧ abortVals′ = abortVals ∪ {s}
  ∧ status′ = [status EXCEPT ![p] = “aborted”]
  ∧ UNCHANGED ⟨pending, initVals, execAcks, panicAcks, dState, accStatus,
    network, pastPending⟩

 The Acceptors

InitAcceptor ≜
    ∧  accStatus = [rep ∈ Acceptor
       ↦ IF Initial THEN “ready” ELSE  “idle”]
    ∧  dState = [rep ∈ Acceptor ↦ Bot]

WakeUp(rep) ≜
    ∧  accStatus[rep] = “idle”
    ∧  accStatus′ = [accStatus EXCEPT ![rep] = “ready”]
    ∧  IF rep = Leader
          THEN ∃ p ∈ P, s ∈ S :
               ∧  RcvSnd(rep, ⟨“init”, s⟩, p,
                    $[a \in Follower \mapsto \{\langle \text{“leaderInit”}, s\rangle\}])$
                $\land\ dState' = [dState$ EXCEPT $![rep] = s]$
            ELSE $\exists\, s \in S :$
                $\land\ Rcv(rep, \langle \text{“leaderInit”}, s\rangle, Leader)$
                $\land\ dState' = [dState$ EXCEPT $![rep] = s]$
        $\land$ UNCHANGED $\langle status, pending, execAcks, interface,$
          $initVals, abortVals, panicAcks, pastPending\rangle$

$Exec(rep) \triangleq$
  $\land\ accStatus[rep] = \text{“ready”}$
  $\land$ IF $rep = Leader$
      THEN $\exists\, p \in P, req \in Req :$
       LET $newDState \triangleq dState[rep] \bullet req$IN
         $\land\ RcvSnd(rep, \langle \text{“req”}, req\rangle, p, [x \in Follower \cup \{p\} \mapsto$
           IF $x \in Follower$
            THEN $\{\langle \text{“leaderExec”}, newDState, p\rangle\}$
            ELSE $\{\langle \text{“execAck”}, dState[rep] \bullet req\rangle\}])$
         $\land\ dState' = [dState$ EXCEPT $![rep] = newDState]$
      ELSE $\exists\, s \in S, p \in P :$
       $\land$    $\exists\, req \in Req : s = dState[rep] \bullet req$  don’t skip updates
       $\land$    $RcvSnd(rep, \langle \text{“leaderExec”}, s, p\rangle, Leader,$
         $[q \in \{p\} \mapsto \{\langle \text{“execAck”}, s\rangle\}])$
       $\land\ dState' = [dState$ EXCEPT $![rep] = s]$
  $\land$ UNCHANGED $\langle status, pending, execAcks, interface, accStatus,$
    $initVals, abortVals, panicAcks, pastPending\rangle$

$Stop(a) \triangleq$
  $\land\ accStatus[a] \in \{\text{“ready”}, \text{“stopped”}\}$
  $\land\ \exists\, p \in P : RcvSnd(a, \langle \text{“panic”}\rangle, p,$
    $[q \in \{p\} \mapsto \{\langle \text{“panicAck”}, dState[a]\rangle\}])$
  $\land\ accStatus' = [accStatus$ EXCEPT $![a] = \text{“stopped”}]$
  $\land$ UNCHANGED $\langle status, initVals, pending, execAcks, interface, dState,$
    $abortVals, panicAcks, pastPending\rangle$

$Timeout(a) \triangleq$
  $\land\ accStatus[a] = \text{“idle”}$
  $\land\ dState' = [dState$ EXCEPT $![a] = None]$
  $\land\ accStatus' = [accStatus$ EXCEPT $![a] = \text{“stopped”}]$
  $\land$ UNCHANGED $\langle status, initVals, pending, execAcks, interface,$
    $abortVals, panicAcks, pastPending, network\rangle$

 The full spec

$Init \triangleq$
  $\land\ InitProcs$
  $\land\ InitAcceptor$
```

```
    $\land\ \mathit{interface} = \mathit{InterfaceInit}$
    $\land\ \mathit{network} = \{\}$
    $\land\ \mathit{abortVals} = \{\}$
    $\land\ \mathit{initVals} = \{\}$
    $\land\ \mathit{pastPending} = \{\}$

$\mathit{Next}\ \triangleq$
    $\lor\ \exists\, p \in P : \mathit{Inv}(p) \lor \mathit{Ini}(p) \lor \mathit{RcvPanicAck}(p) \lor \mathit{RcvExecAcC}(p)$
        $\lor\ \mathit{Panic}(p) \lor \mathit{Abo}(p) \lor \mathit{Res}(p)$
    $\lor\ \exists\, a \in \mathit{Acceptor} : \mathit{WakeUp}(a) \lor \mathit{Exec}(a) \lor \mathit{Stop}(a) \lor \mathit{Timeout}(a)$

$\mathit{Spec}\ \triangleq\ \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$

$\mathit{Safe}\ \triangleq\ \textsc{instance}\ \mathit{SafeMPGC}$

$\textsc{theorem}\ \mathit{Spec} \Rightarrow \mathit{Safe}!\mathit{Spec}$
```

## A.3 Shared-Memory Consensus

```
┌──────────────────── MODULE SharedMemConsensus ────────────────────┐

EXTENDS Library, Consensus, TLCDefs

INSTANCE RDR

 local variables start with an underscore.
VARIABLES
    v, d, contention, pending, pc,
    interface,
    spinterface,
    abortVals  ghost variable

VARIABLES splitterPc, x, y

INSTANCE SpecLinInterface
INSTANCE SplitterConcreteInterface

Splitter ≜ INSTANCE Splitter WITH
  interface ← spinterface,
  pc ← splitterPc

splitterVars ≜ ⟨splitterPc, x, y, spinterface⟩


vars ≜ ⟨v, d, contention, pending, pc, interface, spinterface, abortVals⟩

TypeInvariant ≜
    ∧ pc ∈ [P → {"L1", "L2", "L3", "L4", "L5", "L6", "L7", "L8", "L9",
           "L10", "COMMITTED", "ABORTED"}]
    ∧ pending ∈ [P → Req]
    ∧ v ∈ S
    ∧ d ∈ BOOLEAN
    ∧ contention ∈ BOOLEAN

Init ≜
    ∧ pc = [p ∈ P ↦ "L1"]
    ∧ d = FALSE
    ∧ v = Bot
    ∧ contention = FALSE
    ∧ pending = [p ∈ P ↦ Some(Req)]
    ∧ interface  = InterfaceInit
    ∧ abortVals = {}

PCFromTo(p, l1, l2) ≜
    ∧ pc[p] = l1
    ∧ pc′ = [pc EXCEPT ![p] = l2]

Return(p, o) ≜
```

```
    ∧ pc′ = [pc EXCEPT ![p] = "COMMITTED"]
    ∧ Response(p, o)

GiveUp(p, av) ≜
    ∧ pc′  = [pc EXCEPT ![p] = "ABORTED"]
    ∧ Abort(p, pending[p][2], av)
    ∧ abortVals′ = abortVals ∪ {av}

Step1(p) ≜
    ∧  pc[p] = "L1"
    ∧  pc′ = [pc EXCEPT ![p] = "L2"]
    ∧  ∃ c ∈ C :
           ∧ Invoke(p, c)
           ∧ pending′ = [pending EXCEPT ![p] = ⟨p, c⟩]
    ∧  UNCHANGED ⟨v, d, contention, spinterface, abortVals⟩

Step2(p) ≜
    ∧  pc[p]  = "L2"
    ∧  IF   d = TRUE
         THEN
            IF ¬contention
             THEN
               ∧ Return(p, v)
               ∧ UNCHANGED  abortVals
             ELSE  GiveUp(p, v)
         ELSE
            ∧ pc′ =  [pc EXCEPT ![p] = "L3"]
            ∧ UNCHANGED ⟨interface, abortVals⟩
    ∧  UNCHANGED ⟨v, d, contention, pending, spinterface⟩

Step3a(p) ≜
    ∧ pc[p] = "L3"
    ∧ InvokeSplitter(p, spinterface, spinterface′)
    ∧ UNCHANGED ⟨v, d, contention, pending, pc, interface, abortVals⟩

Step3b(p) ≜
    ∧ ∃ b ∈ BOOLEAN  :
        ∧  SplitterResponse(p, b, spinterface, spinterface′)
        ∧  IF  b
             THEN pc′ = [pc EXCEPT ![p] = "L4"]
             ELSE  pc′ = [pc EXCEPT ![p] = "L9"]
    ∧ UNCHANGED ⟨v, d, contention, pending, interface, abortVals⟩

Step4(p) ≜
    ∧ PCFromTo(p, "L4", "L5")
    ∧ v′ = pending[p][2]
    ∧ UNCHANGED ⟨d, contention, pending, interface, spinterface, abortVals⟩
```

```
$Step5(p) \triangleq$
    $\land pc[p] = \text{"L5"}$
    $\land$ IF $\neg contention$
        THEN
            $\land pc' = [pc$ EXCEPT $![p] = \text{"L6"}]$
        ELSE
            $\land pc' = [pc$ EXCEPT $![p] = \text{"L8"}]$
    $\land$ UNCHANGED $\langle v, d, contention, pending, interface, spinterface, abortVals \rangle$

$Step6(p) \triangleq$
    $\land PCFromTo(p, \text{"L6"}, \text{"L7"})$
    $\land d' =$ TRUE
    $\land$ UNCHANGED $\langle v, contention, pending, interface, spinterface, abortVals \rangle$

$Step7(p) \triangleq$
    $\land pc[p] = \text{"L7"}$
    $\land Return(p, v)$
    $\land$ UNCHANGED $\langle v, d, contention, pending, spinterface, abortVals \rangle$

$Step8(p) \triangleq$
    $\land pc[p] = \text{"L8"}$
    $\land GiveUp(p, Bot)$
    $\land$ UNCHANGED $\langle v, d, contention, pending, spinterface \rangle$

$Step9(p) \triangleq$
    $\land PCFromTo(p, \text{"L9"}, \text{"L10"})$
    $\land contention' =$ TRUE
    $\land$ UNCHANGED $\langle v, d, pending, interface, spinterface, abortVals \rangle$

$Step10(p) \triangleq$
    $\land pc[p] = \text{"L10"}$
    $\land GiveUp(p, v)$
    $\land$ UNCHANGED $\langle v, d, contention, pending, spinterface \rangle$

$Next \triangleq \exists p \in P :$
    $\lor Step1(p) \lor Step2(p) \lor Step3a(p) \lor Step3b(p) \lor Step4(p) \lor Step5(p) \lor Step6(p) \lor Step7(p)$
    $\lor Step8(p) \lor Step9(p) \lor Step10(p)$

$NextComp \triangleq$
  $\land$  $\lor Next$
         $\lor$ UNCHANGED $vars$
  $\land$  $\lor Splitter!Next$
         $\lor x' = x \land y' = y \land splitterPc' = splitterPc \land spinterface' = spinterface$

$Spec \triangleq Init \land Splitter!Init \land \Box[NextComp]_{\langle vars, splitterVars \rangle}$

$status \triangleq$
  $[p \in P \mapsto$
```

```
    IF pc[p] ∈ {"L1", "COMMITTED"}
     THEN "ready"
     ELSE  IF pc[p] = "ABORTED"
             THEN "aborted"
             ELSE "pending"]

dState ≜
  IF ∃ p ∈ P : pc[p] ∈ {"L6", "L7", "COMMITTED"}
   THEN v
   ELSE Bot

SLin ≜ INSTANCE SpecLin WITH
  Initial   ← TRUE,
  pending ← pending,
  initialized ← TRUE,
  initVals ← {}

THEOREM Spec ⇒ SLin!Spec
```

```
──────────── MODULE SplitterConcreteInterface ────────────
EXTENDS Library

CONSTANT P

SpInterfaceType ≜ [
    resp : [P → [
        output : BOOLEAN ,
        flag : BOOLEAN ]],
    inv : [P → BOOLEAN ]]
SpInterfaceInit ≜ [
    resp ↦ [p ∈ P ↦ [
        output ↦ Some(BOOLEAN ),
        flag ↦ Some(BOOLEAN )]],
    inv ↦ [p ∈ P ↦ Some(BOOLEAN )]]

InvokeSplitter(p, interface, newinterface) ≜
    newinterface = [interface EXCEPT !.inv = [@ EXCEPT ![p] = ¬@]]

SplitterResponse(p, b, interface, newinterface) ≜
    newinterface = [interface EXCEPT !.resp = [@ EXCEPT ![p] = [
        output ↦ b,
        flag ↦ ¬@.flag]]]
──────────────────────────────────────────────────────────
```

```
MODULE Splitter

EXTENDS SplitterInterface, Library

CONSTANT P

VARIABLES
    x, y, pc,
    interface

vars ≜ ⟨x, y, pc, interface⟩
Labels ≜ {"START", "L1", "L2", "L3", "L4", "END"}

TypeInvariant ≜
    ∧ x ∈ P
    ∧ y ∈ BOOLEAN
    ∧ pc ∈ [P → Labels]
    ∧ interface ∈ SpInterfaceType

PCFromTo(p, l1, l2) ≜
    ∧ pc[p] = l1
    ∧ pc′ = [pc EXCEPT ![p] = l2]

Start(p) ≜
    ∧ pc[p] = "START"
    ∧ InvokeSplitter(p, interface, interface′)
    ∧ pc′ = [pc EXCEPT ![p] = "L1"]
    ∧ UNCHANGED ⟨x, y⟩

WriteX(p) ≜
    ∧ PCFromTo(p, "L1", "L2")
    ∧ x′ = p
    ∧ interface′ = interface
    ∧ UNCHANGED y
     ∧ UNCHANGED ⟨y, interface⟩

TestY(p) ≜
    ∧ pc[p] = "L2"
    ∧ IF y = TRUE
       THEN  ∧ SplitterResponse(p, FALSE, interface, interface′)
             ∧ pc′ = [pc EXCEPT ![p] = "END"]
       ELSE  ∧ pc′ = [pc EXCEPT ![p] = "L3"]
             ∧ interface′ = interface
    ∧ UNCHANGED ⟨x, y⟩

WriteY(p) ≜
    ∧ PCFromTo(p, "L3", "L4")
    ∧ y′ = TRUE
    $\land\ interface' = interface$
    $\land$ UNCHANGED $x$

$TestX(p)\ \triangleq$
    $\land\ PCFromTo(p,\ \text{“L4”},\ \text{“END”})$
    $\land$ IF $x = p$
        THEN $SplitterResponse(p,\ \text{TRUE},\ interface,\ interface')$
        ELSE $SplitterResponse(p,\ \text{FALSE},\ interface,\ interface')$
    $\land$ UNCHANGED $\langle x,\ y \rangle$

$Init\ \triangleq$
    $\land\ pc = [p \in P \mapsto \text{“START”}]$
    $\land\ x = Some(P)$
    $\land\ y = \text{FALSE}$
    $\land\ interface = SpInterfaceInit$

$Next\ \triangleq\ \exists\, p \in P :$
    $Start(p) \lor WriteX(p) \lor TestY(p) \lor WriteY(p) \lor TestX(p)$

$Spec\ \triangleq\ Init \land \Box[Next]_{vars}$
```

## A.4 Auxiliary Definitions

```
┌──────────────────────── MODULE Library ────────────────────────┐
EXTENDS Sequences, Naturals, FiniteSets

Last(s)   ≜ s[Len(s)]

Some(S) ≜ CHOOSE e ∈ S : TRUE

Min(i, j) ≜ IF i < j THEN i ELSE j
Max(S, LessEq(_, _)) ≜ CHOOSE e ∈ S : ∀ e1 ∈ S : LessEq(e1, e)

Image(f) ≜ {f[x] : x ∈ DOMAIN f}

 Sequences with no duplicates:

RECURSIVE NoDupRec(_, _)
NoDupRec(es, seen) ≜
    IF es = ⟨⟩
     THEN TRUE
     ELSE
        IF es[1] ∈ seen
         THEN FALSE
         ELSE NoDupRec(Tail(es), seen ∪ {es[1]})
NoDup(es) ≜
  NoDupRec(es, {})
NoDupSeq(E) ≜
  {es ∈ Seq(E) : NoDup(es)}

 Removing duplicates from a sequence:

RECURSIVE RemDupRec(_, _)
RemDupRec(es, seen) ≜
  IF es = ⟨⟩
   THEN ⟨⟩
   ELSE
    IF es[1] ∈ seen
     THEN RemDupRec(Tail(es), seen)
     ELSE ⟨es[1]⟩ ∘ RemDupRec(Tail(es), seen ∪ {es[1]})
RemDup(es) ≜ RemDupRec(es, {})

 Sequence prefixes:

Prefix(s1, s2) ≜
    ∧ Len(s1) ≤ Len(s2)
    ∧ ∀ i ∈ DOMAIN s1 : s1[i] = s2[i]

 The longest common prefix of two sequences:

RECURSIVE LongestCommonPrefixLenRec(_, _, _)
```

# B Isabelle/HOL Theories

In this appendix we include our formalization of the theory of I/O automata in Isabelle/HOL, the specification of the $SLin$ family of I/O automata, and the invariants and the refinement mapping used to prove the idempotence of $SLin$. The idempotence proof is described at a high level in section 5.5.

# 1 Sequences as Lists

```
theory Sequences
imports Main
begin

locale Sequences
begin
```

We reverse the order of application of *op* # and *op* @ because it we think that it is easier to think of sequences as growing to the right.

```
no-notation Cons (infixr # 65)
abbreviation Append  (infixl # 65)
  where Append xs x ≡ Cons x xs
no-notation append (infixr @ 65)
abbreviation Concat  (infixl @ 65)
  where Concat xs ys ≡ append ys xs

end

end
```

# 2 I/O Automata with Finite-Trace Semantics

```
theory IOA
imports Main Sequences
begin
```

This theory is inspired by the IOA theory of Olaf Mller

```
locale IOA = Sequences

record 'a signature =
  inputs::'a set
  outputs::'a set
  internals::'a set

context IOA
begin
```

## 2.1 Signatures

```
definition actions :: 'a signature ⇒ 'a set where
  actions asig ≡ inputs asig ∪ outputs asig ∪ internals asig
```

```
definition externals :: 'a signature ⇒ 'a set where
  externals asig ≡ inputs asig ∪ outputs asig

definition locals :: 'a signature ⇒ 'a set where
  locals asig ≡ internals asig ∪ outputs asig

definition is-asig :: 'a signature ⇒ bool where
  is-asig triple ≡
     inputs triple ∩ outputs triple = {} ∧
     outputs triple ∩ internals triple = {} ∧
     inputs triple ∩ internals triple = {}

lemma internal-inter-external:
  assumes is-asig sig
  shows internals sig ∩ externals sig = {}
  using assms by (auto simp add:internals-def externals-def is-asig-def)

definition hide-asig where
  hide-asig asig actns ≡
    (|inputs = inputs asig − actns, outputs = outputs asig − actns,
      internals = internals asig ∪actns|)

end
```

## 2.2 I/O Automata

```
type-synonym
  ('s,'a) transition = 's × 'a × 's

record ('s,'a) ioa =
  asig::'a signature
  start::'s set
  trans::('s,'a)transition set

context IOA
begin

abbreviation act A ≡ actions (asig A)
abbreviation ext A ≡ externals (asig A)
abbreviation int where int A ≡ internals (asig A)
abbreviation inp A ≡ inputs (asig A)
abbreviation out A ≡ outputs (asig A)
abbreviation local A ≡ locals (asig A)
```

```
definition is-ioa::('s,'a) ioa ⇒ bool where
  is-ioa A ≡ is-asig (asig A)
    ∧ (∀ triple ∈ trans A . (fst o snd) triple ∈ act A)

definition hide where
  hide A actns ≡ A(|asig := hide-asig (asig A) actns|)

definition is-trans::'s ⇒ 'a ⇒ ('s,'a)ioa ⇒ 's ⇒ bool where
  is-trans s1 a A s2 ≡ (s1,a,s2) ∈ trans A

notation
  is-trans (- −-−-⟶ - [81,81,81,81] 100)

definition rename-set where
  rename-set A ren ≡ {b. ∃ x ∈ A . ren b = Some x}

definition rename where
rename A ren ≡
  (|asig = (|inputs = rename-set (inp A) ren,
    outputs = rename-set (out A) ren,
    internals = rename-set (int A) ren|),
   start = start A,
   trans = {tr. ∃ x . ren (fst (snd tr)) = Some x ∧ (fst tr) −x−A⟶ (snd (snd
tr))}|)
```

Reachable states and invariants

```
inductive
  reachable :: ('s,'a) ioa ⇒ 's ⇒ bool
  for A :: ('s,'a) ioa
  where
    reachable-0: s ∈ start A ⟹ reachable A s
  | reachable-n: ⟦ reachable A s; s −a−A⟶ t ⟧ ⟹ reachable A t

definition invariant where
  invariant A P ≡ (∀ s . reachable A s ⟶ P(s))

theorem invariantI:
  fixes A P
  assumes ⋀ s . s ∈ start A ⟹ P s
  and ⋀ s t a . ⟦reachable A s; P s; s −a−A⟶ t⟧ ⟹ P t
  shows invariant A P
proof −
  { fix s
    assume reachable A s
    hence P s
    proof (induct rule:reachable.induct)
      fix s
      assume s ∈ start A
      thus P s using assms(1) by simp
    next
      fix a s t
      assume reachable A s and P s and  s −a−A⟶ t
      thus P t using assms(2) by simp
    qed }
  thus ?thesis by (simp add:invariant-def)
qed

end
```

## 2.3 Composition of Families of I/O Automata

```
record ('id, 'a) family =
  ids :: 'id set
  memb :: 'id ⇒ 'a

context IOA
begin

definition is-ioa-fam where
  is-ioa-fam fam ≡ ∀ i ∈ ids fam . is-ioa (memb fam i)

definition compatible2  where
  compatible2 A B ≡
   out A ∩ out B = {} ∧
   int A ∩ act B = {} ∧
   int B ∩ act A = {}


definition compatible::('id, ('s,'a)ioa) family ⇒ bool where
  compatible fam ≡ finite (ids fam) ∧
    (∀ i ∈ ids fam . ∀ j ∈ ids fam . i ≠ j ⟶
      compatible2 (memb fam i) (memb fam j))

definition asig-comp2 where
  asig-comp2 A B ≡
     (|inputs = (inputs A ∪ inputs B) − (outputs A ∪ outputs B),
      outputs = outputs A ∪ outputs B,
      internals = internals A ∪ internals B|)

definition asig-comp::('id, ('s,'a)ioa) family ⇒ 'a signature where
```

```
  asig-comp fam ≡
    (| inputs = ⋃ i∈(ids fam). inp (memb fam i)
         − (⋃ i∈(ids fam). out (memb fam i)),
       outputs = ⋃ i∈(ids fam). out (memb fam i),
       internals = ⋃ i∈(ids fam). int (memb fam i) |)

definition par2 (infixr ∥ 10) where
  A ∥ B ≡
     (|asig = asig-comp2 (asig A) (asig B),
      start = {pr. fst pr ∈ start A ∧ snd pr ∈ start B},
      trans = {tr.
       let s = fst tr; a = fst (snd tr); t = snd (snd tr)
       in (a ∈ act A ∨ a ∈ act B)
          ∧ (if a ∈ act A
             then fst s −a−A⟶ fst t
             else fst s = fst t)
          ∧ (if a ∈ act B
             then snd s −a−B⟶ snd t
             else snd s = snd t) }|)

definition par::('id, ('s,'a)ioa) family ⇒ ('id ⇒ 's,'a)ioa where
  par fam ≡ let ids = ids fam; memb = memb fam in
     (| asig = asig-comp fam,
       start = {s . ∀ i∈ids . s i ∈ start (memb i)},
       trans = { (s, a, s') .
         (∃ i∈ids . a ∈ act (memb i))
         ∧ (∀ i∈ids .
            if a ∈ act (memb i)
            then s i −a−(memb i)⟶ s' i
            else s i = (s' i)) } |)

lemmas asig-simps = hide-asig-def is-asig-def locals-def externals-def actions-def
  hide-def compatible-def asig-comp-def
lemmas ioa-simps = rename-def rename-set-def is-trans-def is-ioa-def par-def

end
```

## 2.4 Executions and Traces

```
type-synonym
  ('s,'a)pairs = ('a×'s) list
type-synonym
  ('s,'a)execution = 's × ('s,'a)pairs
type-synonym
  'a trace = 'a list
```

```
record ('s,'a)execution-module =
  execs::('s,'a)execution set
  asig::'a signature

record 'a trace-module =
  traces::'a trace set
  asig::'a signature

context IOA
begin

fun is-exec-frag-of::('s,'a)ioa ⇒ ('s,'a)execution ⇒ bool where
  is-exec-frag-of A (s,(ps#p')#p) =
    (snd p' −fst p−A⟶ snd p ∧ is-exec-frag-of A (s, (ps#p')))
| is-exec-frag-of A (s, [p]) = s −fst p−A⟶ snd p
| is-exec-frag-of A (s, []) = True

definition is-exec-of::('s,'a)ioa ⇒ ('s,'a)execution ⇒ bool where
  is-exec-of A e ≡ fst e ∈ start A ∧ is-exec-frag-of A e

definition filter-act where
  filter-act ≡ map fst

definition schedule where
  schedule ≡ filter-act o snd

definition trace where
  trace sig ≡ filter (λ a . a ∈ externals sig) o schedule



definition is-schedule-of where
  is-schedule-of A sch ≡
    (∃ e . is-exec-of A e ∧ sch = filter-act (snd e))

definition is-trace-of where
  is-trace-of A tr ≡
    (∃ sch . is-schedule-of A sch ∧ tr = filter (λ a. a ∈ ext A) sch)

definition traces where
  traces A ≡ {tr. is-trace-of A tr}

lemma traces-alt:
  shows traces A = {tr . ∃ e . is-exec-of A e
  ∧ tr = trace (ioa.asig A) e}
proof −
  { fix t
    assume a:t ∈ traces A
    have ∃ e . is-exec-of A e ∧ trace (ioa.asig A) e = t
    proof −
      from a obtain sch where 1:is-schedule-of A sch
        and 2:t = filter (λ a. a ∈ ext A) sch
        by (auto simp add:traces-def is-trace-of-def)
      from 1 obtain e where 3:is-exec-of A e and 4:sch = filter-act (snd e)
        by (auto simp add:is-schedule-of-def)
      from 4 and 2 have trace (ioa.asig A) e = t
        by (simp add:trace-def schedule-def)
      with 3 show ?thesis by fast
    qed }
  moreover
  { fix e
    assume is-exec-of A e
    hence trace (ioa.asig A) e ∈ traces A
      by (auto simp add:trace-def schedule-def traces-def
          is-trace-of-def is-schedule-of-def is-exec-of-def)
        (metis (full-types) pair-collapse) }
  ultimately show ?thesis by blast
qed

lemmas trace-simps = traces-def is-trace-of-def is-schedule-of-def filter-act-def is-exec-of-def
  trace-def schedule-def

definition proj-trace::'a trace ⇒ ('a signature) ⇒ 'a trace (infixr | 12) where
  proj-trace t sig ≡ filter (λ a . a ∈ actions sig) t

definition ioa-implements :: ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool   (infixr =<| 12)
where
  A =<| B ≡ inp A = inp B ∧ out A = out B ∧ traces A ⊆ traces B
```

## 2.5 Operations on Executions

```
definition cons-exec where
  cons-exec e p ≡ (fst e, (snd e)#p)

definition append-exec where
  append-exec e e' ≡ (fst e, (snd e)@(snd e'))

fun last-state where
  last-state (s,[]) = s
```

```
| last-state (s,ps#p) = snd p

lemma last-state-reachable:
  fixes A e
  assumes is-exec-of A e
  shows reachable A (last-state e) using assms
proof –
  have is-exec-of A e ⟹ reachable A (last-state e)
  proof (induction snd e arbitrary: e)
    case Nil
    from Nil.prems have 1:fst e ∈ start A by (simp add:is-exec-of-def)
   from Nil.hyps have 2:last-state e = fst e by (metis last-state.simps(1) surjective-pairing)
    from 1 and 2 and Nil.hyps show ?case by (metis reachable-0)
  next
    case (Cons p ps e)
    let ?e′ = (fst e, ps)
    have ih:reachable A (last-state ?e′)
    proof –
      from Cons.prems and Cons.hyps(2) have is-exec-of A ?e′
        by (simp add:is-exec-of-def) (metis is-exec-frag-of.simps(1,3) list.exhaust
pair-collapse)
      with Cons.hyps(1) show ?thesis by auto
    qed
    from Cons.prems and Cons.hyps(2) have (last-state ?e′)–(fst p)–A⟶(snd
p)
      by (simp add:is-exec-of-def) (cases (A,fst e,ps#p) rule:is-exec-frag-of.cases,
auto)
    with ih and Cons.hyps(2) show ?case
      by (metis last-state.simps(2) reachable.simps surjective-pairing)
  qed
  thus ?thesis using assms by fastforce
qed

lemma trans-from-last-state:
  assumes is-exec-frag-of A e and (last-state e)–a–A⟶s′
  shows is-exec-frag-of A (cons-exec e (a,s′))
    using assms by (cases (A, fst e, snd e) rule:is-exec-frag-of.cases, auto simp
add:cons-exec-def)

lemma exec-frag-prefix:
  fixes A p ps
  assumes is-exec-frag-of A (cons-exec e p)
  shows is-exec-frag-of A e
    using assms by (cases (A, fst e, snd e) rule:is-exec-frag-of.cases, auto simp
add:cons-exec-def)
```

```
lemma trace-same-ext:
  fixes A B e
  assumes ext A = ext B
  shows trace (ioa.asig A) e = trace (ioa.asig B) e
  using assms by (auto simp add:trace-def)

lemma trace-append-is-append-trace:
  fixes e e' sig
  shows trace sig (append-exec e' e) = trace sig e' @ trace sig e
  by (simp add:append-exec-def trace-def schedule-def filter-act-def)

lemma append-exec-frags-is-exec-frag:
  fixes e e' A as
  assumes is-exec-frag-of A e and last-state e = fst e'
  and is-exec-frag-of A e'
  shows is-exec-frag-of A (append-exec e e')
proof −
  from assms show ?thesis
  proof (induct (fst e',snd e') arbitrary:e' rule:is-exec-frag-of.induct)
    case (3 A)
    from 3.hyps and 3.prems(1)
    show ?case by (simp add:append-exec-def)
  next
    case (2 A p)
    have last-state e −(fst p)−A−→ snd p using 2.prems(2,3) and 2.hyps
      by (metis is-exec-frag-of.simps(2) pair-collapse)
    hence is-exec-frag-of A (fst e, (snd e)#p) using 2.prems(1)
      by (metis cons-exec-def pair-collapse trans-from-last-state)
    moreover
    have append-exec e e' = (fst e, (snd e)#p) using 2.hyps
      by (metis append-Cons append-Nil append-exec-def)
    ultimately
    show ?case by auto
  next
    case (1 A ps p' p e')
    have is-exec-frag-of A (fst e, (snd e)@((ps#p')#p))
    proof −
      have is-exec-frag-of A (fst e, (snd e)@(ps#p'))
        by (metis 1.hyps 1.prems append-exec-def cons-exec-def
            exec-frag-prefix fst-conv prod-eqI snd-conv)
      moreover
      have snd p' −(fst p)−A−→ snd p using 1.prems(3) 1.hyps(2)
        by (metis is-exec-frag-of.simps(1) pair-collapse)
      ultimately show ?thesis by simp
    qed
    moreover have append-exec e e′ = (fst e, (snd e)@((ps#p′)#p))
      by (metis 1.hyps(2) append-exec-def)
    ultimately show ?case by simp
  qed
qed

lemma last-state-of-append:
  fixes e e′
  assumes fst e′ = last-state e
  shows last-state (append-exec e e′) = last-state e′
  using assms by (cases e′ rule:last-state.cases, auto simp add:append-exec-def)


end

end
```

# 3 Definition and Soundness of Refinement Mappings, Forward Simulations and Backward Simulations

```
theory Simulations
imports IOA
begin

context IOA
begin

definition refines where
  refines e s a t A f ≡ fst e = f s ∧ last-state e = f t ∧ is-exec-frag-of A e
           ∧ (let tr = trace (ioa.asig A) e in
               if a ∈ ext A then tr = [a] else tr = [])
definition
  is-ref-map :: ('s1 ⇒ 's2) ⇒ ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool where
  is-ref-map f B A ≡
    (∀ s ∈ start B . f s ∈ start A) ∧ (∀ s t a. reachable B s ∧ s −a−B⟶ t
      ⟶ (∃ e . refines e s a t A f ))

definition
  is-forward-sim :: ('s1 ⇒ ('s2 set)) ⇒ ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool where
  is-forward-sim f B A ≡
   (∀ s ∈ start B . f s ∩ start A ≠ {})
   ∧ (∀ s s′ t a. s′ ∈ f s ∧ s −a−B⟶ t ∧ reachable B s
       ⟶ (∃ e . fst e = s′ ∧ last-state e ∈ f t ∧ is-exec-frag-of A e
            ∧ (let tr = trace (ioa.asig A) e in
                 if a ∈ ext A then tr = [a] else tr = [])))

definition
  is-backward-sim :: ('s1 ⇒ ('s2 set)) ⇒ ('s1,'a)ioa ⇒ ('s2,'a)ioa ⇒ bool where
  is-backward-sim f B A ≡
   (∀ s . f s ≠ {}) (* Restricting this to reachable states would suffice *)
   ∧ (∀ s ∈ start B . f s ⊆ start A)
   ∧ (∀ s t a t′. t′ ∈ f t ∧ s −a−B⟶ t ∧ reachable B s
       ⟶ (∃ e . fst e ∈ f s ∧ last-state e = t′ ∧ is-exec-frag-of A e
             ∧ (let tr = trace (ioa.asig A) e in
                  if a ∈ ext A then tr = [a] else tr = [])))
```

## 3.1 A series of lemmas that will be useful in the soundness proofs

```
lemma step-eq-traces:
  fixes e-B′ A e e-A′ a t
  defines e-A ≡ append-exec e-A′ e and e-B ≡ cons-exec e-B′ (a,t)
  and tr ≡ trace (ioa.asig A) e
  assumes 1:trace (ioa.asig A) e-A′ = trace (ioa.asig A) e-B′
  and 2:if a ∈ ext A then tr = [a] else tr = []
  shows trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof −
  have 3:trace (ioa.asig A) e-B =
       (if a ∈ ext A then (trace (ioa.asig A) e-B′) # a else trace (ioa.asig A) e-B′)
    using e-B-def by (simp add:trace-def schedule-def filter-act-def cons-exec-def)
  have 4:trace (ioa.asig A) e-A =
        (if a ∈ ext A then trace (ioa.asig A) e-A′ # a else trace (ioa.asig A) e-A′)
    using 2 trace-append-is-append-trace[of ioa.asig A e-A′ e]
      by(auto simp add:e-A-def tr-def split add:split-if-asm)
  show ?thesis using 1 3 4 by simp
qed

lemma exec-inc-imp-trace-inc:
  fixes A B
  assumes ext B = ext A
  and ⋀ e-B . is-exec-of B e-B
    ⟹ ∃ e-A . is-exec-of A e-A ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
  shows traces B ⊆ traces A
proof −
```

```
  { fix t
    assume t ∈ traces B
    with this obtain e where 1:t = trace (ioa.asig B) e and 2:is-exec-of B e
      using traces-alt assms(1) by blast
    from 1 and assms(1) have 3:t = trace (ioa.asig A) e by (simp add:trace-def)
    from 2 3 and assms(2) obtain e′ where
      is-exec-of A e′ ∧ trace (ioa.asig A) e′ = trace (ioa.asig A) e by blast
    hence t ∈ traces A using 3 traces-alt by fastforce }
  thus ?thesis by fast
qed
```

## 3.2 Soundness of Refinement Mappings

```
lemma ref-map-execs:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA and e-B
  assumes is-ref-map f B A and is-exec-of B e-B
  shows ∃ e-A . is-exec-of A e-A
    ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof −
  note assms(2)
  hence ∃ e-A . is-exec-of A e-A
                ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
                ∧ last-state e-A = f (last-state e-B)
  proof (induction snd e-B arbitrary:e-B)
    case Nil
    let ?e-A = (f (fst e-B), [])
    have ⋀ s . s ∈ start B ⟹ f s ∈ start A using assms(1) by (simp add:is-ref-map-def)
    hence is-exec-of A ?e-A using Nil.prems(1) by (simp add:is-exec-of-def)
    moreover
    have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
      by (simp add:trace-simps) (metis Nil.hyps filter.simps(1) map.simps(1))
    moreover
    have last-state ?e-A = f (last-state e-B)
      using Nil.hyps by (metis last-state.simps(1) pair-collapse)
    ultimately show ?case by fast
  next
    case (Cons p ps e-B)
    let ?e-B′ = (fst e-B, ps)
    let ?s = last-state ?e-B′ let ?t = snd p let ?a = fst p
    have 1:is-exec-of B ?e-B′ and 2:?s−?a−B⟶?t
      using Cons.prems and Cons.hyps(2)
        by (simp-all add:is-exec-of-def,
            cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto,
            cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto)
    with Cons.hyps(1) obtain e-A′ where ih1:is-exec-of A e-A′
```

```
      and ih2:trace (ioa.asig A) e-A' = trace (ioa.asig A) ?e-B'
      and ih3:last-state e-A' = f ?s by fastforce
    from 1 have 3:reachable B ?s using last-state-reachable by fast
    obtain e where 4:fst e = f ?s and 5:last-state e = f ?t
    and 6:is-exec-frag-of A e
    and 7:let tr = trace (ioa.asig A) e in if ?a ∈ ext A
      then tr = [?a] else tr = []
        using 2 and 3 and assms(1)
          by (force simp add:is-ref-map-def refines-def)
    let ?e-A = append-exec e-A' e
    have is-exec-of A ?e-A
      using ih1 ih3 4 6 append-exec-frags-is-exec-frag[of A e e-A']
        by (metis append-exec-def append-exec-frags-is-exec-frag
          fst-conv is-exec-of-def)
    moreover
    have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
      using ih2 Cons.hyps(2) 7 step-eq-traces[of A e-A' ?e-B' ?a e]
        by (auto simp add:cons-exec-def) (metis pair-collapse)
    moreover have last-state ?e-A = f ?t using ih3 4 5 last-state-of-append
      by metis
    ultimately show ?case using Cons.hyps(2)
      by (metis last-state.simps(2) surjective-pairing)
  qed
  thus ?thesis by blast
qed

theorem ref-map-soundness:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA
  assumes is-ref-map f B A and ext A = ext B
  shows traces B ⊆ traces A
  using assms ref-map-execs exec-inc-imp-trace-inc by metis
```

## 3.3 Soundness of Forward Simulations

```
lemma forward-sim-execs:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA set and e-B
  assumes is-forward-sim f B A and is-exec-of B e-B
  shows ∃ e-A . is-exec-of A e-A
    ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof –
  note assms(2)
  hence ∃ e-A . is-exec-of A e-A
              ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
              ∧ last-state e-A ∈ f (last-state e-B)
  proof (induction snd e-B arbitrary:e-B)
  case Nil
  have ⋀ s . s ∈ start B ⟹ f s ∩ start A ≠ {}
    using assms(1) by (simp add:is-forward-sim-def)
  with this obtain s′ where 1:s′ ∈ f (fst e-B) and 2:s′ ∈ start A
    by (metis Int-iff Nil.prems all-not-in-conv is-exec-of-def)
  let ?e-A = (s′, [])
  have is-exec-of A ?e-A using 2 by (simp add:is-exec-of-def)
  moreover
  have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B using Nil.hyps
    by (simp add:trace-def schedule-def filter-act-def)
  moreover
  have last-state ?e-A ∈ f (last-state e-B)
    using Nil.hyps 1 by (metis last-state.simps(1) surjective-pairing)
  ultimately show ?case by fast
next
  case (Cons p ps e-B)
  let ?e-B′ = (fst e-B, ps)
  let ?s = last-state ?e-B′ let ?t = snd p let ?a = fst p
  have 1:is-exec-of B ?e-B′ and 2:?s−?a−B⟶?t
    using Cons.prems and Cons.hyps(2)
      by (simp-all add:is-exec-of-def,
          cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto,
          cases (B,fst e-B,ps#p) rule:is-exec-frag-of.cases, auto)
  with Cons.hyps(1) obtain e-A′ where ih1:is-exec-of A e-A′
    and ih2:trace (ioa.asig A) e-A′ = trace (ioa.asig A) ?e-B′
    and ih3:last-state e-A′ ∈ f ?s by fastforce
  from 1 have 3:reachable B ?s using last-state-reachable by fast
  obtain e where 4:fst e = last-state e-A′ and 5:last-state e ∈ f ?t
  and 6:is-exec-frag-of A e
  and 7:let tr = trace (ioa.asig A) e in if ?a ∈ ext A then tr = [?a] else tr = []
    using 2 3 assms(1) ih3 by (simp add:is-forward-sim-def)
      (metis pair-collapse prod.inject)
  let ?e-A = append-exec e-A′ e
  have is-exec-of A ?e-A
    using ih1 ih3 4 6 append-exec-frags-is-exec-frag[of A e e-A′]
      by (metis append-exec-def append-exec-frags-is-exec-frag
          fst-conv is-exec-of-def)
  moreover
  have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
    using ih2 Cons.hyps(2) 7 step-eq-traces[of A e-A′ ?e-B′ ?a e]
      by (auto simp add:cons-exec-def Let-def) (metis pair-collapse)
  moreover have last-state ?e-A ∈ f ?t using ih3 4 5 last-state-of-append
    by metis
  ultimately show ?case using Cons.hyps(2)
    by (metis last-state.simps(2) surjective-pairing)
  qed
  thus ?thesis by blast
qed

theorem forward-sim-soundness:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA set
  assumes is-forward-sim f B A and ext A = ext B
  shows traces B ⊆ traces A
  using assms forward-sim-execs exec-inc-imp-trace-inc by metis
```

### 3.4 Soundness of Backward Simulations

```
lemma backward-sim-execs:
  fixes A::('sA,'a)ioa and B::('sB,'a)ioa and f::'sB ⇒ 'sA set and e-B
  assumes is-backward-sim f B A and is-exec-of B e-B
  shows ∃ e-A . is-exec-of A e-A
    ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
proof −
  note assms(2)
  hence ∀ s ∈ f (last-state e-B). ∃ e-A .
          is-exec-of A e-A
          ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
          ∧ last-state e-A = s
  proof (induction snd e-B arbitrary:e-B)
    case Nil
    { fix s' assume 1:s' ∈ f(last-state e-B)
      have 2:⋀ s . s ∈ start B ⟹ f s ⊆ start A
        using assms(1) by (simp add:is-backward-sim-def)
      from Nil 1 2 have 3:s' ∈ start A
      by (metis (full-types) is-exec-of-def last-state.simps(1) set-mp surjective-pairing)
      let ?e-A = (s', [])
      have 4:is-exec-of A ?e-A using 3 by (simp add:is-exec-of-def)
      have 5:trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B using Nil.hyps
        by (simp add:trace-def schedule-def filter-act-def)
      have 6:last-state ?e-A ∈ f (last-state e-B)
        using Nil.hyps 1 by (metis last-state.simps(1))
      note 4 5 6 }
    thus ?case by fastforce
  next
    case (Cons p ps e-B)
    { fix t' assume 8:t' ∈ f (last-state e-B)
      let ?e-B' = (fst e-B, ps)
      let ?s = last-state ?e-B' let ?t = snd p let ?a = fst p
      have 5:?t = last-state e-B using Cons.hyps(2)
        by (metis last-state.simps(2) pair-collapse)
      have 1:is-exec-of B ?e-B′ and 2:?s−?a−B⟶?t
        using Cons.prems and Cons.hyps(2)
          by (simp-all add:is-exec-of-def,
              cases (B,fst e-B, ps#p) rule:is-exec-frag-of.cases, auto,
              cases (B,fst e-B, ps#p) rule:is-exec-frag-of.cases, auto)
      from 1 have 3:reachable B ?s using last-state-reachable by fast
      obtain e where 4:fst e ∈ f ?s and 5:last-state e = t′
      and 6:is-exec-frag-of A e
      and 7:let tr = trace (ioa.asig A) e in
        if ?a ∈ ext A then tr = [?a] else tr = []
        using 2 assms(1) 8 5 3 by (auto simp add: is-backward-sim-def, metis)
      obtain e-A′ where ih1:is-exec-of A e-A′
        and ih2:trace (ioa.asig A) e-A′ = trace (ioa.asig A) ?e-B′
        and ih3:last-state e-A′ = fst e
          using 1 4 Cons.hyps(1) by (metis snd-conv)
      let ?e-A = append-exec e-A′ e
      have is-exec-of A ?e-A
        using ih1 ih3 4 6 append-exec-frags-is-exec-frag[of A e e-A′]
          by (metis append-exec-def append-exec-frags-is-exec-frag
            fst-conv is-exec-of-def)
      moreover
      have trace (ioa.asig A) ?e-A = trace (ioa.asig A) e-B
        using ih2 Cons.hyps(2) 7 step-eq-traces[of A e-A′ ?e-B′ ?a e]
          by (auto simp add:cons-exec-def Let-def) (metis pair-collapse)
      moreover have last-state ?e-A = t′ using ih3 5 last-state-of-append
        by metis
      ultimately have ∃ e-A . is-exec-of A e-A
        ∧ trace (ioa.asig A) e-A = trace (ioa.asig A) e-B
        ∧ last-state e-A = t′ by blast }
    thus ?case by blast
  qed
  moreover
  from assms(1) have total:⋀ s . f s ≠ {} by (simp add:is-backward-sim-def)
  ultimately show ?thesis by fast
qed

theorem backward-sim-soundness:
  fixes A::(′sA,′a)ioa and B::(′sB,′a)ioa and f::′sB ⇒ ′sA set
  assumes is-backward-sim f B A and ext A = ext B
  shows traces B ⊆ traces A
  using assms backward-sim-execs exec-inc-imp-trace-inc by metis

end

end
```

# 4 Recoverable Data Types

```
theory RDR
imports Main Sequences
begin
```

## 4.1 The pre-RDR locale

```
locale pre-RDR = Sequences +
  fixes δ::'a ⇒ ('b × 'c) ⇒ 'a (infix · 65)
  and γ::'a ⇒ ('b × 'c) ⇒ 'd
  and bot::'a (⊥)
begin

fun exec::'a ⇒ ('b×'c)list ⇒ 'a (infix ⋆ 65) where
  exec s Nil = s
| exec s (rs#r) = (exec s rs) · r

definition less-eq (infix ⪯ 50) where
  less-eq s s' ≡ ∃ rs . s' = (s⋆rs)

definition less (infix ≺ 50) where
  less s s' ≡ less-eq s s' ∧ s ≠ s'

definition is-lb where
  is-lb s s1 s2 ≡ s ⪯ s2 ∧ s ⪯ s1

definition is-glb where
  is-glb s s1 s2 ≡ is-lb s s1 s2 ∧ (∀ s' . is-lb s' s1 s2 ⟶ s' ⪯ s)

definition contains where
  contains s r ≡ ∃ rs . r ∈ set rs ∧ s = (⊥ ⋆ rs)

definition inf  (infix ⊓ 65) where
  inf s1 s2 ≡ THE s . is-glb s s1 s2
```

## 4.2 Useful Lemmas in the pre-RDR locale

```
lemma exec-cons:
  s ⋆ (rs # r)= (s ⋆ rs) · r by simp

lemma exec-append:
  (s ⋆ rs) ⋆ rs'  = s ⋆ (rs@rs')
```

```
proof (induct rs')
  show (s ⋆ rs) ⋆ []  = s ⋆ (rs@[]) by simp
next
  fix rs' r
  assume ih:(s ⋆ rs) ⋆ rs'  = s ⋆ (rs@rs')
  thus (s ⋆ rs) ⋆ (rs'#r)  = s ⋆ (rs @ (rs'#r))
    by (metis append-Cons exec-cons)
qed

lemma trans:
  assumes s1 ⪯ s2 and s2 ⪯ s3
  shows s1 ⪯ s3 using assms
    by (auto simp add:less-eq-def, metis exec-append)

lemma contains-star:
  fixes s r rs
  assumes contains s r
  shows contains (s ⋆ rs) r
proof (induct rs)
  case Nil
  show contains (s ⋆ []) r using assms by auto
next
  case (Cons r' rs)
  with this obtain rs' where 1:s ⋆ rs = ⊥ ⋆ rs' and 2:r ∈ set rs'
    by (auto simp add:contains-def)
  have 3:s ⋆ (rs#r') = ⊥⋆(rs'#r') using 1 by fastforce
  show contains (s ⋆ (rs#r')) r using 2 3
    by (auto simp add:contains-def) (metis exec-cons set-rev-mp set-subset-Cons)
qed

lemma preceq-star: s ⋆ (rs#r) ⪯ s' ⟹ s ⋆ rs ⪯ s'
by (metis pre-RDR.exec.simps(1) pre-RDR.exec.simps(2) pre-RDR.less-eq-def trans)

end
```

## 4.3 The RDR locale

```
locale RDR = pre-RDR +
  assumes idem1:contains s r ⟹ s · r = s
  and idem2:⋀ s r r' . fst r ≠ fst r' ⟹ γ s r = γ ((s · r) · r') r
  and antisym:⋀ s1 s2 . s1 ⪯ s2 ∧ s2 ⪯ s1 ⟹ s1 = s2
  and glb-exists:⋀ s1 s2 . ∃ s . is-glb s s1 s2
  and consistency:⋀s1 s2 s3 rs . s1 ⪯ s2 ⟹ s2 ⪯ s3 ⟹ s3 = s1 ⋆ rs
    ⟹ ∃ rs' rs'' . s2 = s1 ⋆ rs' ∧ s3 = s2 ⋆ rs''
      ∧ set rs' ⊆ set rs ∧ set rs'' ⊆ set rs
```

```
 and bot:⋀ s . ⊥ ≼ s

begin

lemma inf-glb:is-glb (s1 ⊓ s2) s1 s2
proof –
 { fix s s′
   assume is-glb s s1 s2 and is-glb s′ s1 s2
   hence s = s′ using antisym by (auto simp add:is-glb-def is-lb-def) }
   from this and glb-exists show ?thesis
     by (auto simp add:inf-def, metis (lifting) theI′)
qed

sublocale ordering less-eq less
proof
 fix s
 show s ≼ s
 by (metis exec.simps(1) less-eq-def)
next
 fix s s′
 show s ≺ s′ = (s ≼ s′ ∧ s ≠ s′)
 by (auto simp add:less-def)
next
 fix s s′
 assume s ≼ s′ and s′ ≼ s
 thus s = s′
 using antisym by auto
next
 fix s1 s2 s3
 assume s1 ≼ s2 and s2 ≼ s3
 thus s1 ≼ s3
 using trans by blast
qed

sublocale semilattice-set inf
proof
 fix s
 show s ⊓ s = s
   using inf-glb
   by (metis antisym is-glb-def is-lb-def refl)
next
 fix s1 s2
 show s1 ⊓ s2 = (s2 ⊓ s1)
   using inf-glb
```

```
  by (smt antisym is-glb-def pre-RDR.is-lb-def)
next
 fix s1 s2 s3
 show (s1 ⊓ s2) ⊓ s3 = (s1 ⊓ (s2 ⊓ s3))
  using inf-glb
  by(auto simp add:is-glb-def is-lb-def, smt antisym trans)
qed

sublocale semilattice-order-set inf less-eq less
proof
 fix s s′
 show s ≼ s′ = (s = s ⊓ s′)
 by (metis antisym idem inf-glb pre-RDR.is-glb-def pre-RDR.is-lb-def)
next
 fix s s′
 show s ≺ s′ = (s ≼ s′ ∧ s ≠ s′)
 by (auto simp add:less-def)
qed

notation F (∏ - [99])
```

## 4.4 Some useful lemmas

```
lemma idem-star:
fixes r s rs
assumes contains s r
shows s ⋆ rs = s ⋆ (filter (λ x . x ≠ r) rs)
proof (induct rs)
 case Nil
 show s ⋆ [] = s ⋆ (filter (λ x . x ≠ r) [])
  using assms by auto
next
 case (Cons r′ rs)
 have 1:contains (s ⋆ rs) r using assms and contains-star by auto
 show s ⋆ (rs#r′) = s ⋆ (filter (λ x . x ≠ r) (rs#r′))
 proof (cases r′ = r)
  case True
  hence s ⋆ (rs#r′) = s ⋆ rs using idem1 1 by auto
  thus ?thesis using Cons by simp
 next
  case False
  thus ?thesis using Cons by auto
 qed
qed
```

```
lemma idem-star2:
  fixes s rs'
  shows ∃ rs' . s ⋆ rs = s ⋆ rs' ∧ set rs' ⊆ set rs
    ∧ (∀ r ∈ set rs' . ¬ contains s r)
proof (induct rs)
  case Nil
  thus ∃ rs' . s ⋆ [] = s ⋆ rs' ∧ set rs' ⊆ set []
    ∧ (∀ r ∈ set rs' . ¬ contains s r) by force
next
  case (Cons r rs)
  obtain rs' where 1:s ⋆ rs = s ⋆ rs' and 2:set rs' ⊆ set rs
  and 3:∀ r ∈ set rs' . ¬ contains s r using Cons(1) by blast
  show ∃ rs' . s ⋆ (rs#r) = s ⋆ rs' ∧ set rs' ⊆ set (rs#r)
    ∧ (∀ r ∈ set rs' . ¬ contains s r)
  proof (cases contains s r)
    case True
    have s⋆(rs#r) = s⋆rs'
    proof −
      have s ⋆ (rs#r) = s⋆rs using True
        by (metis contains-star exec-cons idem1)
      moreover
      have s ⋆ (rs'#r) = s⋆rs' using True
        by (metis contains-star exec-cons idem1)
      ultimately show ?thesis using 1 by simp
    qed
    moreover have set rs' ⊆ set (rs#r) using 2
      by (simp, metis subset-insertI2)
    moreover have ∀ r ∈ set rs' . ¬ contains s r
      using 3 by assumption
    ultimately show ?thesis by blast
  next
    case False
    have s⋆(rs#r) = s⋆(rs'#r) using 1 by simp
    moreover
    have set (rs'#r) ⊆ set (rs#r) using 2 by auto
    moreover have ∀ r ∈ set (rs'#r) . ¬ contains s r
      using 3 False by auto
    ultimately show ?thesis by blast
  qed
qed

lemma idem2-star:
assumes contains s r
and ⋀ r' . r' ∈ set rs ⟹ fst r' ≠ fst r
shows γ s r = γ (s ⋆ rs) r using assms
```

```
proof (induct rs)
  case Nil
  show γ s r = γ (s ⋆ []) r by simp
next
  case (Cons r′ rs)
  thus γ s r = γ (s ⋆ (rs#r′)) r
    using assms by auto
      (metis contains-star fst-conv idem1 idem2 prod.exhaust)
qed
```

```
lemma glb-common:
fixes s1 s2 s rs1 rs2
assumes s1 = s ⋆ rs1 and s2 = s ⋆ rs2
shows ∃ rs . s1 ⊓ s2 = s ⋆ rs ∧ set rs ⊆ set rs1 ∪ set rs2
proof −
  have 1:s ≼ s1 and 2:s ≼ s2 using assms by (auto simp add:less-eq-def)
  hence 3:s ≼ s1 ⊓ s2 by (metis inf-glb is-lb-def pre-RDR.is-glb-def)
  have 4:s1 ⊓ s2 ≼ s1 by (metis cobounded1)
  show ?thesis using 3 4 assms(1) and consistency by blast
qed
```

```
lemma glb-common-set:
fixes ss s0 rset
assumes finite ss  and ss ≠ {}
and ⋀ s . s ∈ ss ⟹ ∃ rs . s = s0 ⋆ rs ∧ set rs ⊆ rset
shows ∃ rs . ⨅ ss = s0 ⋆ rs ∧ set rs ⊆ rset
using assms
proof (induct ss rule:finite-ne-induct)
  case (singleton s)
  obtain rs where s = s0 ⋆ rs ∧ set rs ⊆ rset using singleton by force
  moreover have ⨅ {s} = s using singleton by auto
  ultimately show ∃ rs . ⨅ {s} = s0 ⋆ rs ∧ set rs ⊆ rset by blast
next
  case (insert s ss)
  have 1:⋀ s′ . s′ ∈ ss ⟹ ∃ rs . s′ = s0 ⋆ rs ∧ set rs ⊆ rset
    using insert(5) by force
  obtain rs where 2:⨅ ss = s0 ⋆ rs and 3:set rs ⊆ rset
    using insert(4) 1 by blast
  obtain rs′ where 4:s = s0 ⋆ rs′and 5:set rs′ ⊆ rset
    using insert(5) by blast
  have 6:⨅ (insert s ss) = s ⊓ (⨅ ss)
    by (metis insert.hyps(1−3) insert-not-elem)
  obtain rs″ where 7:⨅ (insert s ss) = s0 ⋆ rs″
```

```
    and 8:set rs'' ⊆ set rs' ∪ set rs
    using glb-common 2 4 6 by force
  have 9:set rs'' ⊆ rset using 3 5 8 by blast
  show ∃ rs . ⨅ (insert s ss) = s0 ⋆ rs ∧ set rs ⊆ rset
    using 7 9 by blast
qed

end

end
```

# 5 The SLin Automata specification

```
theory SLin
imports IOA RDR
begin

datatype ('a,'b,'c,'d)SLin-action =
— The nat component is the instance number
  Invoke nat 'b 'c
| Response nat 'b 'd
| Switch nat 'b 'c 'a
| Recover nat
| Linearize nat

datatype SLin-status = Sleep | Pending | Ready | Aborted

record ('a,'b,'c)SLin-state =
  pending :: 'b ⇒ 'b × 'c
  initVals :: 'a set
  abortVals :: 'a set
  status :: 'b ⇒ SLin-status
  dstate :: 'a
  initialized :: bool

locale SLin = RDR + IOA
begin

definition
  asig :: nat ⇒ nat ⇒ ('a,'b,'c,'d)SLin-action signature
  — The first instance has number 0
  where
  asig i j ≡ ⦇
```

```
    inputs = {act . ∃ p c iv i' .
      (i ≤ i' ∧ i' < j ∧ act = Invoke i' p c) ∨ (i > 0 ∧ act = Switch i p c iv)},
    outputs = {act . ∃ p c av i' outp .
      (i ≤ i' ∧ i' < j ∧ act = Response i' p outp) ∨ act = Switch j p c av},
    internals = {act. ∃ i' . i ≤ i' ∧ i' < j
      ∧ (act = Linearize i' ∨ act = Recover i')} ⦈

definition pendingReqs :: ('a,'b,'c)SLin-state ⇒ ('b×'c) set
  where
  pendingReqs s ≡ {r . ∃ p .
     r = pending s p
     (*∧ ¬ contains (dstate s) r*)
     ∧ status s p ∈ {Pending, Aborted}}

definition Inv :: 'b ⇒ 'c
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Inv p c s s' ≡
    status s p = Ready
    ∧ s' = s⦇pending := (pending s)(p := (p,c)),
        status := (status s)(p := Pending)⦈

definition pendingSeqs where
  pendingSeqs s ≡ {rs . set rs ⊆ pendingReqs s}

definition Lin :: ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Lin s s' ≡ ∃ rs ∈ pendingSeqs s .
      initialized s
      ∧ (∀ av ∈ abortVals s . (dstate s) ⋆ rs ≼ av)
      ∧ s' = s⦇dstate := (dstate s) ⋆ rs⦈

definition initSets where
  initSets s ≡ {ivs . ivs ≠ {} ∧ ivs ⊆ initVals s}

definition safeInits where
  safeInits s ≡ if initVals s = {} then {}
    else {d . ∃ ivs ∈ initSets s . ∃ rs ∈ pendingSeqs s .
      d = ⨅ivs ⋆ rs ∧ (∀ av ∈ abortVals s . d ≼ av)}

definition initAborts where
  initAborts s ≡ { d .dstate s ≼ d
    ∧ ((∃ rs ∈ pendingSeqs s . d = dstate s ⋆ rs)
      ∨ (∃ ivs ∈ initSets s . dstate s ≼ ⨅ivs
        ∧ (∃ rs ∈ pendingSeqs s . d = ⨅ivs ⋆ rs))) }
```

```
definition uninitAborts where
  uninitAborts s ≡ { d .
    ∃ ivs ∈ initSets s . ∃ rs ∈ pendingSeqs s .
      d = ⨅ ivs ⋆ rs }

definition safeAborts::('a,'b,'c)SLin-state ⇒ 'a set where
  safeAborts s ≡ if initialized s then initAborts s
    else uninitAborts s

definition Reco :: ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Reco s s' ≡
      (∃ p . status s p ≠ Sleep)
      ∧ ¬ initialized s
      ∧ (∃ d ∈ safeInits s .
        s' = s⦇dstate := d, initialized := True⦈)

definition Resp :: 'b ⇒ 'd ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Resp p ou s s' ≡
      status s p = Pending
      ∧ initialized s
      ∧ contains (dstate s) (pending s p)
      ∧ ou = γ (dstate s) (pending s p)
      ∧ s' = s ⦇status := (status s)(p := Ready)⦈

definition Init :: 'b ⇒ 'c ⇒ 'a
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Init p c iv s s' ≡
    status s p = Sleep
    ∧ s' = s ⦇initVals := {iv} ∪ (initVals s),
        status := (status s)(p := Pending),
        pending := (pending s)(p := (p,c))⦈

definition Abort :: 'b ⇒ 'c ⇒ 'a
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Abort p c av s s' ≡
     status s p = Pending ∧ pending s p = (p,c)
     ∧ av ∈ safeAborts s
     ∧ s' = s⦇status := (status s)(p := Aborted),
      abortVals := (abortVals s ∪ {av})⦈

definition trans where
```

$\mathit{trans}\ i\ j \equiv \{\ (s,a,s')\ .\ \mathit{case}\ a\ \mathit{of}$
$\quad \mathit{Invoke}\ i'\ p\ c \Rightarrow i \leq i' \wedge i < j \wedge \mathit{Inv}\ p\ c\ s\ s'$
$\mid \mathit{Response}\ i'\ p\ \mathit{ou} \Rightarrow i \leq i' \wedge i < j \wedge \mathit{Resp}\ p\ \mathit{ou}\ s\ s'$
$\mid \mathit{Switch}\ i'\ p\ c\ v \Rightarrow (i > \mathit{0} \wedge i' = i \wedge \mathit{Init}\ p\ c\ v\ s\ s')$
$\qquad \vee\ (i' = j \wedge \mathit{Abort}\ p\ c\ v\ s\ s')$
$\mid \mathit{Linearize}\ i' \Rightarrow i' = i \wedge \mathit{Lin}\ s\ s'$
$\mid \mathit{Recover}\ i' \Rightarrow i > \mathit{0} \wedge i' = i \wedge \mathit{Reco}\ s\ s'\ \}$

**definition** *start* **where**
$\quad \mathit{start}\ i \equiv \{\ s\ .$
$\qquad \forall\ p\ .\ \mathit{status}\ s\ p = (\mathit{if}\ i > \mathit{0}\ \mathit{then}\ \mathit{Sleep}\ \mathit{else}\ \mathit{Ready})$
$\qquad \wedge\ \mathit{dstate}\ s = \bot$
$\qquad \wedge\ (\mathit{if}\ i > \mathit{0}\ \mathit{then}\ \neg\ \mathit{initialized}\ s\ \mathit{else}\ \mathit{initialized}\ s)$
$\qquad \wedge\ \mathit{initVals}\ s = \{\}$
$\qquad \wedge\ \mathit{abortVals}\ s = \{\}\}$

**definition** *ioa* **where**
$\quad \mathit{ioa}\ i\ j \equiv$
$\qquad (\!|\mathit{ioa.asig} = \mathit{asig}\ i\ j\ ,$
$\qquad\ \ \mathit{start} = \mathit{start}\ i,$
$\qquad\ \ \mathit{trans} = \mathit{trans}\ i\ j|\!)$

**end**

**end**

# 6 The Consensus Data Type

**theory** *Consensus*
**imports** *RDR*
**begin**

This theory provides a model for the RDR locale, thus showing that the assumption of the RDR locale are consistent.

**typedecl** *proc*
**typedecl** *val*

**locale** *Consensus*
— To avoid name clashes
**begin**

**fun** $\delta{::}\mathit{val\ option} \Rightarrow (\mathit{proc} \times \mathit{val}) \Rightarrow \mathit{val\ option}$ (**infix** $\cdot$ *65*) **where**
$\quad \delta\ \mathit{None}\ r = \mathit{Some}\ (\mathit{snd}\ r)$

```
| δ (Some v) r = Some v

fun γ::val option ⇒ (proc × val) ⇒ val where
  γ None r = snd r
| γ (Some v) r = v

interpretation pre-RDR δ γ None .
notation exec (infix ⋆ 65)
notation less-eq (infix ≼ 50 )
notation None (⊥)

lemma single-use:
  fixes r rs
  shows  ⊥ ⋆ ([r]@rs) = Some (snd r)
proof (induct rs)
  case Nil
  thus ?case by simp
next
  case (Cons r rs)
  thus ?case by auto
qed

lemma bot: ∃ rs . s = ⊥ ⋆ rs
proof (cases s)
  case None
  hence s = ⊥ ⋆ [] by auto
  thus ?thesis by blast
next
  case (Some v)
  obtain r where ⊥ ⋆ [r] = Some v by force
  thus ?thesis using Some by metis
qed

lemma prec-eq-None-or-equal:
fixes s1 s2
assumes s1 ≼ s2
shows s1 = None ∨ s1 = s2 using assms single-use
proof −
  { assume 1:s1 ≠ None and 2:s1 ≠ s2
    obtain r rs where 3:s1 = ⊥ ⋆ ([r]@rs) using bot using 1
      by (metis append-butlast-last-id pre-RDR.exec.simps(1))
    obtain rs' where 4:s2 = s1 ⋆ rs' using assms
      by (auto simp add:less-eq-def)
    have s2 = ⊥ ⋆ ([r]@(rs@rs')) using 3 4
      by (metis exec-append)
    hence s1 = s2 using 3
      by (metis single-use)
    with 2 have False by auto }
  thus ?thesis by blast
qed

interpretation RDR δ γ ⊥
proof (unfold-locales)
  fix s r
  assume contains s r
  show s • r = s
  proof −
    obtain rs where s = ⊥ ⋆ rs and rs ≠ []
      using ‹contains s r›
      by (auto simp add:contains-def, force)
    thus ?thesis
    by (metis δ.simps(2) rev-exhaust single-use)
  qed
next
  fix s and r r' :: proc × val
  assume 1:fst r ≠ fst r'
  thus γ s r = γ ((s • r) • r') r
    by (metis δ.simps γ.simps not-Some-eq)
next
  fix s1 s2
  assume s1 ≼ s2 ∧ s2 ≼ s1
  thus s1 = s2 by (metis prec-eq-None-or-equal)
next
  fix s1 s2
  show ∃ s . is-glb s s1 s2
  by (simp add:is-glb-def is-lb-def)
    (metis bot pre-RDR.less-eq-def prec-eq-None-or-equal)
next
  fix s
  show ⊥ ≼ s
  by (metis bot pre-RDR.less-eq-def)
next
  fix s1 s2 s3 rs
  assume s1 ≼ s2 and s2 ≼ s3 and s3 = s1 ⋆ rs
  thus ∃ rs' rs'' . s2 = s1 ⋆ rs' ∧ s3 = s2 ⋆ rs''
    ∧ set rs' ⊆ set rs ∧ set rs'' ⊆ set rs
    by (metis Consensus.prec-eq-None-or-equal
      in-set-insert insert-Nil list.distinct(1)
        pre-RDR.exec.simps(1) subsetI)
qed
```

```
end

end
```

# 7 Idempotence of the SLin I/O automaton

```
theory Idempotence
imports SLin Simulations
begin

locale Idempotence = SLin +
  fixes id1 id2 :: nat
  assumes id1:0 < id1 and id2:id1 < id2
begin

lemmas ids = id1 id2

definition composition where
  composition ≡
     hide ((ioa 0 id1) ‖ (ioa id1 id2))
          {act . EX p c av . act = Switch id1 p c av }

lemmas comp-simps = hide-def composition-def ioa-def par2-def is-trans-def
  start-def actions-def asig-def trans-def

lemmas trans-defs = Inv-def Lin-def Resp-def Init-def
  Abort-def Reco-def

declare split-if-asm [split]
```

## 7.1 A case rule for decomposing the transition relation of the composition of two SLins

```
declare comp-simps [simp]
lemma trans-elim:
  fixes  s t a s' t' P
  assumes (s,t) −a−composition⟶ (s',t')
  obtains
    (Invoke1) i p c
      where Inv p c s s' ∧ t = t'
      and i < id1 and a = Invoke i p c
  | (Invoke2) i p c
```

```
    where Inv p c t t′ ∧ s = s′
    and id1 ≤ i ∧ i < id2 and a = Invoke i p c
  | (Switch1) p c av
    where Abort p c av s s′ ∧ Init p c av t t′
    and a = Switch id1 p c av
  | (Switch2) p c av
    where s = s′ ∧ Abort p c av t t′
    and a = Switch id2 p c av
  | (Response1) i p ou
    where Resp p ou s s′∧ t = t′
    and i < id1 and a = Response i p ou
  | (Response2) i p ou
    where Resp p ou t t′ ∧ s = s′
    and id1 ≤ i ∧ i < id2 and a = Response i p ou
  | (Lin1) Lin s s′ ∧ t = t′ and a = Linearize 0
  | (Lin2) Lin t t′ ∧ s = s′ and a = Linearize id1
  | (Reco2) Reco t t′ ∧ s = s′ and a = Recover id1
declare comp-simps [simp del]
```

## 7.2 Definition of the Refinement Mapping

```
fun f :: (('a,'b,'c)SLin-state ∗ ('a,'b,'c)SLin-state) ⇒ ('a,'b,'c)SLin-state
  where
  f (s1, s2) =
    ⦇pending = λ p. (if status s1 p ≠ Aborted then pending s1 p else pending s2 p),
     initVals = {},
     abortVals = abortVals s2,
     status = λ p. (if status s1 p ≠ Aborted then status s1 p else status s2 p),
     dstate = (if dstate s2 = ⊥ then dstate s1 else dstate s2),
     initialized = True⦈
```

## 7.3 Invariants

```
declare
  trans-defs [simp]

fun P1 where
  P1 (s1,s2) = (∀ p . status s1 p ∈ {Pending, Aborted}
    ⟶ fst (pending s1 p) = p)

fun P2 where
  P2 (s1,s2) = (∀ p . status s2 p ≠ Sleep ⟶ fst (pending s2 p) = p)
```

**fun** *P3* **where**
$P3\ (s1,s2) = (\forall\ p\ .\ (status\ s2\ p = Ready \longrightarrow initialized\ s2))$

**fun** *P4* **where**
$P4\ (s1,s2) = ((\forall\ p\ .\ status\ s2\ p = Sleep) = (initVals\ s2 = \{\}))$

**fun** *P5* **where**
$P5\ (s1,s2) = (\forall\ p\ .\ status\ s1\ p \neq Sleep \land initialized\ s1\ \land initVals\ s1 = \{\})$

**fun** *P6* **where**
$P6\ (s1,s2) = (\forall\ p\ .\ (status\ s1\ p \neq Aborted) = (status\ s2\ p = Sleep))$

**fun** *P7* **where**
$P7\ (s1,s2) = (\forall\ c\ .\ status\ s1\ c = Aborted \land \neg\ initialized\ s2$
$\longrightarrow (pending\ s2\ c = pending\ s1\ c \land status\ s2\ c \in \{Pending,\ Aborted\}))$

**fun** *P8* **where**
$P8\ (s1,s2) = (\forall\ iv \in initVals\ s2\ .\ \exists\ rs \in pendingSeqs\ s1\ .$
$iv = dstate\ s1 \star rs)$

**fun** *P8a* **where**
$P8a\ (s1,s2) = (\forall\ ivs \in initSets\ s2\ .\ \exists\ rs \in pendingSeqs\ s1\ .$
$\bigsqcap ivs = dstate\ s1 \star rs)$

**fun** *P9* **where**
$P9\ (s1,s2) = (initialized\ s2 \longrightarrow dstate\ s1 \preceq dstate\ s2)$

**fun** *P10* **where**
$P10\ (s1,s2) = ((\neg\ initialized\ s2) \longrightarrow (dstate\ s2 = \bot))$

**fun** *P11* **where**
$P11\ (s1,s2) = (initVals\ s2 = abortVals\ s1)$

**fun** *P12* **where**
$P12\ (s1,s2) = (initialized\ s2 \longrightarrow \bigsqcap\ (initVals\ s2) \preceq dstate\ s2)$

**fun** *P13* **where**
$P13\ (s1,s2) = (finite\ (initVals\ s2)$
$\land\ finite\ (abortVals\ s1) \land finite\ (abortVals\ s2))$

**fun** *P14* **where**
$P14\ (s1,s2) = (initialized\ s2 \longrightarrow initVals\ s2 \neq \{\})$

**fun** *P15* **where**
$\mathit{P15}\ (\mathit{s1},\mathit{s2}) = (\forall\ \mathit{av} \in \mathit{abortVals}\ \mathit{s1}\ .\ \mathit{dstate}\ \mathit{s1} \preceq \mathit{av})$

**fun** *P16* **where**
$\mathit{P16}\ (\mathit{s1},\mathit{s2}) = (\mathit{dstate}\ \mathit{s2} \neq \bot \longrightarrow \mathit{initialized}\ \mathit{s2})$

**fun** *P17* **where**
— For the Response1 case of the refinement proof, in case a response is produced in the first instance and the second instance is already initialized
$\mathit{P17}\ (\mathit{s1},\mathit{s2}) = (\mathit{initialized}\ \mathit{s2}$
$\longrightarrow (\forall\ p\ .$
$((\mathit{status}\ \mathit{s1}\ p = \mathit{Ready}$
$\vee\ (\mathit{status}\ \mathit{s1}\ p = \mathit{Pending} \wedge \mathit{contains}\ (\mathit{dstate}\ \mathit{s1})\ (\mathit{pending}\ \mathit{s1}\ p)))$
$\longrightarrow (\exists\ \mathit{rs}\ .\ \mathit{dstate}\ \mathit{s2} = \mathit{dstate}\ \mathit{s1} \star \mathit{rs} \wedge (\forall\ r \in \mathit{set}\ \mathit{rs}\ .\ \mathit{fst}\ r \neq p)))$
$\wedge\ ((\mathit{status}\ \mathit{s1}\ p = \mathit{Pending} \wedge \neg\ \mathit{contains}\ (\mathit{dstate}\ \mathit{s1})\ (\mathit{pending}\ \mathit{s1}\ p))$
$\longrightarrow (\exists\ \mathit{rs}\ .\ \mathit{dstate}\ \mathit{s2} = \mathit{dstate}\ \mathit{s1} \star \mathit{rs} \wedge (\forall\ r \in \mathit{set}\ \mathit{rs}\ .$
$\mathit{fst}\ r = p \longrightarrow r = \mathit{pending}\ \mathit{s1}\ p)))))$

**fun** *P18* **where**
$\mathit{P18}\ (\mathit{s1},\mathit{s2}) = (\mathit{abortVals}\ \mathit{s2} \neq \{\} \longrightarrow (\exists\ p\ .\ \mathit{status}\ \mathit{s2}\ p \neq \mathit{Sleep}))$

**fun** *P19* **where**
$\mathit{P19}\ (\mathit{s1},\mathit{s2}) = (\mathit{abortVals}\ \mathit{s2} \neq \{\} \longrightarrow \mathit{abortVals}\ \mathit{s1} \neq \{\})$

**fun** *P20* **where**
$\mathit{P20}\ (\mathit{s1},\mathit{s2}) = (\forall\ \mathit{av} \in \mathit{abortVals}\ \mathit{s2}\ .\ \mathit{dstate}\ \mathit{s2} \preceq \mathit{av})$

**fun** *P21* **where**
$\mathit{P21}\ (\mathit{s1},\mathit{s2}) = (\forall\ \mathit{av} \in \mathit{abortVals}\ \mathit{s2}\ .\ \bigsqcap(\mathit{abortVals}\ \mathit{s1}) \preceq \mathit{av})$

**fun** *P22* **where**
$\mathit{P22}\ (\mathit{s1},\mathit{s2}) = (\mathit{initialized}\ \mathit{s2} \longrightarrow \mathit{dstate}\ (f\ (\mathit{s1},\mathit{s2})) = \mathit{dstate}\ \mathit{s2})$

**fun** *P23* **where**
$\mathit{P23}\ (\mathit{s1},\mathit{s2}) = ((\neg\ \mathit{initialized}\ \mathit{s2}) \longrightarrow$
$\mathit{pendingSeqs}\ \mathit{s1} \subseteq \mathit{pendingSeqs}\ (f\ (\mathit{s1},\mathit{s2})))$

**fun** *P25* **where**
$\mathit{P25}\ (\mathit{s1},\mathit{s2}) = (\forall\ \mathit{ivs}\ .\ (\mathit{ivs} \in \mathit{initSets}\ \mathit{s2} \wedge \mathit{initialized}\ \mathit{s2}$
$\wedge\ \mathit{dstate}\ \mathit{s2} \preceq \bigsqcap \mathit{ivs})$
$\longrightarrow (\exists\ \mathit{rs}' \in \mathit{pendingSeqs}\ (f\ (\mathit{s1},\mathit{s2}))\ .\ \bigsqcap \mathit{ivs} = \mathit{dstate}\ \mathit{s2} \star \mathit{rs}'))$

```
fun P26 where
  P26 (s1,s2) = (∀ p . (status s1 p = Aborted
    ∧ ¬ contains (dstate s2) (pending s1 p))
      ⟶ (status s2 p ∈ {Pending,Aborted}
        ∧ pending s1 p = pending s2 p))

lemma P1-invariant:
shows invariant (composition) P1
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P1 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P1 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P1 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P2-invariant:
shows invariant (composition) P2
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P2 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P2 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P2 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed


lemma P16-invariant:
shows invariant (composition) P16
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P16 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P16 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P16 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed
```

```
lemma P3-invariant:
shows invariant (composition) P3
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P3 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P3 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P3 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P4-invariant:
shows invariant (composition) P4
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P4 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P4 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P4 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P5-invariant:
shows invariant (composition) P5
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P5 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P5 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P5 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed


lemma P13-invariant:
shows invariant (composition) P13
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P13 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P13 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P13 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, auto)
qed

lemma P20-invariant:
shows invariant (composition) P20
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P20 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P20 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P16:P16 (s1,s2) using P16-invariant and ids
    by (metis IOA.invariant-def)
  show P20 (t1,t2) using trans and hyp and P16
  by (cases rule:trans-elim, auto simp add:safeInits-def safeAborts-def
    initAborts-def uninitAborts-def bot)
qed

lemma P18-invariant:
shows invariant (composition) P18
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P18 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P18 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P18 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, auto)
qed

lemma P14-invariant:
shows invariant (composition) P14
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P14 (s1,s2) using ids by (auto simp add:comp-simps)
```

```
next
  fix s1 s2 t1 t2 a
  assume hyp: P14 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P14 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, auto simp add:safeInits-def)
qed

lemma P15-invariant:
shows invariant (composition) P15
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P15 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P15 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P5:P5 (s1,s2) using P5-invariant and ids
    by (metis IOA.invariant-def)
  show P15 (t1,t2) using trans and hyp and P5
  by (cases rule:trans-elim,
    auto simp add:less-eq-def safeAborts-def initAborts-def)
qed

lemma P6-invariant:
shows invariant (composition) P6
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P6 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P6 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  show P6 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, force+)
qed

lemma P7-invariant:
shows invariant (composition) P7
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P7 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P7 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P7 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P10-invariant:
shows invariant (composition) P10
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P10 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P10 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P10 (t1,t2) using trans and hyp
  by (cases rule:trans-elim) auto
qed

lemma P11-invariant:
shows invariant (composition) P11
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P11 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P11 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  show P11 (t1,t2) using trans and hyp
  by (cases rule:trans-elim, force+)
qed

lemma P8-invariant:
shows invariant (composition) P8
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P8 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P8 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P5:P5 (s1,s2) using P5-invariant and ids
    by (metis IOA.invariant-def)
  from reach have P1:P1 (s1,s2) using P1-invariant and ids
    by (metis IOA.invariant-def)
```

```
from reach have P11:P11 (s1,s2) using P11-invariant and ids
  by (metis IOA.invariant-def)
show P8 (t1,t2) using trans and hyp
proof (cases rule:trans-elim)
  case (Invoke1 i p c)
  assume P8 (s1,s2)
  have pendingSeqs s1 ⊆ pendingSeqs t1
  proof −
    have pending t1 = (pending s1)(p := (p,c))
    and status t1 = (status s1)(p := Pending)
    and status s1 p = Ready
    using Invoke1(1) by auto
   hence pendingReqs s1 ⊆ pendingReqs t1 by (simp add:pendingReqs-def) force
    thus ?thesis by (auto simp add:pendingSeqs-def)
  qed
  moreover have initVals t2 = initVals s2 and dstate t1 = dstate s1
    using Invoke1(1) by auto
  ultimately show P8 (t1,t2) using ‹P8 (s1,s2)› by fastforce
next
  case Lin1
  assume P8 (s1,s2)
  show P8 (t1,t2)
  proof (simp, rule ballI)
    fix iv
    assume 0:iv ∈ initVals t2
    have 1:iv ∈ initVals s2 using Lin1(1) 0 by simp
    have 4:iv ∈ abortVals s1 using 1 P11 by simp
    obtain rs where 2:rs ∈ pendingSeqs s1 and 3:iv = dstate s1 ⋆ rs
      using ‹P8 (s1,s2)› 1 by auto
    obtain rs' where 6:dstate t1 = dstate s1 ⋆ rs' and 5:dstate s1 ⋆ rs' ⪯ iv
      using Lin1(1) 1 4 by auto
    obtain rs'' where 7:iv = (dstate s1 ⋆ rs') ⋆ rs'' and 8:set rs'' ⊆ set rs
      using consistency 3 5 6 by simp (metis less-eq-def)
    have 10:rs'' ∈ pendingSeqs t1
    proof −
      have 9:pendingSeqs t1 = pendingSeqs s1
        using Lin1(1) by (auto simp add:pendingSeqs-def pendingReqs-def)
      thus ?thesis using 8 2  by (auto simp add:pendingSeqs-def)
    qed
    show ∃ rs ∈ pendingSeqs t1 . iv = dstate t1 ⋆ rs
      using 7 10 6 by auto
  qed
next
  case (Response1 i p ou)
  assume ih:P8 (s1,s2)
```

```
    show P8 (t1,t2)
    proof auto
      fix iv
      assume 1:iv ∈ initVals t2
      obtain rs where 2:iv = dstate t1 ⋆ rs and 3:rs ∈ pendingSeqs s1
        using 1 Response1(1) ih by auto
      have 4:pendingReqs t1 = ((pendingReqs s1) − {pending s1 p})
      proof −
        have pending t1 = pending s1 and status t1 = (status s1)(p := Ready)
          and 5:status s1 p = Pending
          using Response1(1) by auto
        moreover have ⋀ q . q ≠ p ⟹ status s1 q ∈ {Pending,Aborted}
          ⟹ pending s1 q ≠ pending s1 p
          using P1 5 by (metis P1.simps insertI1)
        ultimately show ?thesis by (simp add:pendingReqs-def) fastforce
      qed
      have 8:contains (dstate t1) (pending s1 p) using Response1(1) by simp
      def rs′ ≡ filter (λ x . x ≠ (pending s1 p)) rs
      have 9:rs′ ∈ pendingSeqs t1
      proof −
        have 9:pending s1 p ∉ set rs′ by (auto simp add:rs′-def)
        have 10:rs′ ∈ pendingSeqs s1
          using 3 by (auto simp add:rs′-def)
            (metis filter-is-subset mem-Collect-eq pendingSeqs-def subset-trans)
        show ?thesis using 10 9 4 by (auto simp add:pendingSeqs-def)
      qed
      have 10:iv = dstate t1 ⋆ rs′ using 8 2 idem-star rs′-def by fast
      show ∃ rs ∈ pendingSeqs t1 . iv = dstate t1 ⋆ rs using 10 9 by auto
    qed
  next
    case (Switch1 p c av)
    assume P8 (s1,s2)
    have 1:initialized s1 ∧ initVals s1 = {} using P5 by auto
    obtain av where 2:initVals t2 = initVals s2 ∪ {av} and 3:av ∈ safeAborts
s1
      using Switch1(1) by auto
    obtain rs where 4:rs ∈ pendingSeqs s1 and 5:av = dstate s1 ⋆ rs
      using 1 3 by (auto simp add:safeAborts-def initAborts-def initSets-def)
    have 6:dstate s1 = dstate t1 using Switch1(1) by simp
    have 7:pendingSeqs t1 = pendingSeqs s1
    proof −
      have pendingReqs t1 = pendingReqs s1
        using Switch1(1) by (simp add:pendingReqs-def) fastforce
      thus ?thesis by (auto simp add:pendingSeqs-def)
    qed
    show P8 (t1,t2) using ‹P8 (s1,s2)› 2 4 5 6 7 by auto
  next
    case (Invoke2 i p c)
    assume P8 (s1,s2)
    thus P8 (t1,t2) using Invoke2(1) by force
  next
    case Lin2
    assume P8 (s1,s2)
    thus P8 (t1,t2) using Lin2(1) by force
  next
    case (Response2 i p ou)
    assume P8 (s1,s2)
    thus P8 (t1,t2) using Response2(1) by force
  next
    case (Switch2 p c av)
    assume P8 (s1,s2)
    thus P8 (t1,t2) using Switch2(1) by force
  next
    case Reco2
    assume P8 (s1,s2)
    thus P8 (t1,t2) using Reco2(1) by force
  qed
qed

lemma P8a-invariant:
shows invariant (composition) P8a
proof (auto simp:invariant-def)
  fix s1 s2 ivs
  assume 1:reachable (composition) (s1,s2)
  and 2:ivs ∈ initSets s2
  have 3:finite ivs ∧ ivs ≠ {}
  proof −
    have P13 (s1,s2) using P13-invariant assms 1
      by (metis IOA.invariant-def)
    thus ?thesis using 2 finite-subset by (auto simp add:initSets-def)
  qed
  have 4:∀ av ∈ ivs . ∃ rs ∈ pendingSeqs s1 . av = dstate s1 ⋆ rs
  proof −
    have P8:P8 (s1,s2) using P8-invariant assms 1
      by (metis IOA.invariant-def)
    thus ?thesis using 2 by (auto simp add:initSets-def)
  qed
  show ∃ rs ∈ pendingSeqs s1 . ⨅ivs = dstate s1 ⋆ rs
    using 3 4 glb-common-set by (simp add:pendingSeqs-def, metis)
qed
```

```
lemma P12-invariant:
shows invariant (composition) P12
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P12 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P12 (s1,s2) and trans:(s1,s2) −a−composition−→ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  from reach have P13:P13 (s1,s2) using P13-invariant and assms
    by (metis IOA.invariant-def)
  from reach have P14:P14 (s1,s2) using P14-invariant and assms
    by (metis IOA.invariant-def)
  show P12 (t1,t2) using assms and trans and hyp
  proof (cases rule:trans-elim)
    case (Invoke1 i p c)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Invoke1(1) by auto
  next
    case Lin1
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Lin1(1) by auto
  next
    case (Response1 i p ou)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Response1(1) by auto
  next
    case (Switch1 p c av)
    assume ih:P12 (s1,s2)
    have initialized s2 ⟹ ⨅ (initVals t2) ≼ ⨅ (initVals s2)
    proof −
      assume 1:initialized s2
      have initVals t2 = initVals s2 ∪ {av} using Switch1(1) by simp
      hence ⨅ (initVals t2) = ⨅ (initVals s2) ⊓ av
        using insert-not-elem P13 P14 1
          by (metis P13.simps P14.simps Un-empty-right Un-insert-right commute
insert)
      thus ?thesis by (metis cobounded1)
    qed
    moreover have dstate t2 = dstate s2 and initialized s2 = initialized t2
      using Switch1(1) by auto
    ultimately show P12 (t1,t2) using ih by auto (metis absorb2 coboundedI1)
  next
```

```
    case (Invoke2 i p c)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Invoke2(1) by force
  next
    case Lin2
    assume P12 (s1,s2)
    moreover
    have initVals t2 = initVals s2 and initialized s2
    and initialized t2 using Lin2(1) by auto
    moreover
    have dstate s2 ⪯ dstate t2 using Lin2(1) by auto (metis less-eq-def)
    ultimately show P12 (t1,t2) by auto (metis strict-iff-order strict-trans1)
  next
    case (Response2 i p ou)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Response2(1) by force
  next
    case (Switch2 p c av)
    assume P12 (s1,s2)
    thus P12 (t1,t2) using Switch2(1) by force
  next
    case Reco2
    obtain d where 1:d ∈ safeInits s2 and 2:dstate t2 = d
      using Reco2(1) by force
    obtain ivs where 3:ivs ⊆ initVals s2 and 4:ivs ≠ {}
      and 5:⨅ivs ⪯ d
      using 1 by (auto simp add:safeInits-def initSets-def)
        (metis equals0D less-eq-def)
    have 6:⨅ (initVals s2) ⪯ ⨅ ivs using 3 P13 4
      by (metis P13.simps antimono)
    have 7:initVals s2 = initVals t2 using Reco2(1) by auto
    show P12 (t1,t2) using 2 5 6 7
      by (metis P12.simps absorb2 coboundedI1)
  qed
qed

lemma P19-invariant:
shows invariant (composition) P19
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  have P4:P4 (s1,s2) using P4-invariant assms 1
    by (simp add:invariant-def)
  moreover
  have P18:P18 (s1,s2) using P18-invariant assms 1
```

```
    by (metis IOA.invariant-def)
  moreover
  have P11:P11 (s1,s2) using P11-invariant assms 1
    by (metis IOA.invariant-def)
  moreover
  ultimately show P19 (s1,s2) by auto
qed

lemma P9-invariant:
shows invariant (composition) P9
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  have P12:P12 (s1,s2) using P12-invariant assms 1
    by (simp add:invariant-def)
  have P15:P15 (s1,s2) using P15-invariant assms 1
    by (metis IOA.invariant-def)
  have P13:P13 (s1,s2) using P13-invariant assms 1
    by (metis IOA.invariant-def)
  have P14:P14 (s1,s2) using P14-invariant assms 1
    by (metis IOA.invariant-def)
  have P11:P11 (s1,s2) using P11-invariant assms 1
    by (metis IOA.invariant-def)
  have initialized s2 ⟹ dstate s1 ⪯ ⨅(abortVals s1)
    using P13 P15 P14 P11 boundedI by simp
  thus P9 (s1,s2) using P12 P11 by simp (metis trans)
qed

lemma P17-invariant:
shows invariant (composition) P17
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P17 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P17 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach:reachable (composition) (s1,s2)
  show P17 (t1,t2) using trans and hyp
  proof (cases rule:trans-elim)
    case (Invoke1 i p c)
    assume P17 (s1,s2)
    thus P17 (t1,t2) using Invoke1(1) by fastforce
  next
    case (Response1 i p ou)
  assume P17 (s1,s2)
  thus P17 (t1,t2) using Response1(1) by auto
next
  case (Switch1 p c av)
  assume P17 (s1,s2)
  thus P17 (t1,t2) using Switch1(1) by auto
next
  case (Invoke2 i p c)
  assume P17 (s1,s2)
  thus P17 (t1,t2) using Invoke2(1) by force
next
  case (Response2 i p ou)
  assume P17 (s1,s2)
  thus P17 (t1,t2) using Response2(1) by force
next
  case (Switch2 p c av)
  assume P17 (s1,s2)
  thus P17 (t1,t2) using Switch2(1) by force
next
  case Lin1
  assume 1:P17 (s1,s2)
  obtain rs' where 2:dstate t1 = dstate s1 ⋆ rs'
    using Lin1(1) 1 by auto
  have 3:dstate s2 = dstate t2 using Lin1(1) by auto
  have 4:initialized t2 ⟹ dstate t1 ⪯ dstate t2
  proof −
    assume initialized t2
    moreover
    have P9 (t1,t2) using reach trans P9-invariant assms
      by (metis IOA.invariant-def reachable-n)
    ultimately show ?thesis by auto
  qed
  show P17 (t1,t2)
  proof(simp, auto)
    fix p
    assume 5:initialized t2 and 6:status t1 p = Ready
    obtain rs where 7:∀ r ∈ set rs . fst r ≠ p
      and 8:dstate t2 = dstate s1 ⋆ rs
    proof −
      obtain rs where dstate s2 = dstate s1 ⋆ rs
        ∧ (∀ r ∈ set rs . fst r ≠ p) using 1 5 6 Lin1(1) by force
      hence ∀ r ∈ set rs . fst r ≠ p and dstate t2 = dstate s1 ⋆ rs
        using Lin1(1) by auto
      thus ?thesis using that by blast
    qed
  have 9:dstate t1 ≼ dstate t2 using 4 5 by auto
  obtain rs'' where 10:dstate t2 = dstate t1 ⋆ rs''
    and 11:set rs'' ⊆ set rs
      using consistency 2 8 9 by simp (metis less-eq-def)
  have 12:∀ r ∈ set rs'' . fst r ≠ p using 7 11 by blast
  thus ∃ rs . dstate t2 = dstate t1 ⋆ rs ∧ (∀ r ∈ set rs . fst r ≠ p)
    using 10 12 by auto
next
  fix p
  assume 5:initialized t2 and 6:status t1 p = Pending
    and 7:¬ contains (dstate t1) (pending t1 p)
  obtain rs where 8:∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
    and 9:dstate t2 = dstate s1 ⋆ rs
  proof −
    have 9:¬ contains (dstate s1) (pending s1 p)
      using 7 Lin1(1) contains-star by fastforce
    obtain rs where dstate s2 = dstate s1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p)
        using 1 5 6 9 Lin1(1) by force
    hence ∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
      and dstate t2 = dstate s1 ⋆ rs
        using Lin1(1) by auto
    thus ?thesis using that by blast
  qed
  have 10:dstate t1 ≼ dstate t2 using 4 5 by auto
  obtain rs'' where 11:dstate t2 = dstate t1 ⋆ rs''
    and 12:set rs'' ⊆ set rs
      using consistency 2 9 10 by simp (metis less-eq-def)
  have 13:∀ r ∈ set rs'' . fst r = p ⟶ r = pending s1 p
    using 8 12 by blast
  show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending t1 p)
    using 11 13 Lin1(1) by auto
next
  fix p
  assume 5:initialized t2 and 6:status t1 p = Pending
    and 7:contains (dstate t1) (pending t1 p)
  show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r ≠ p)
  proof (cases contains (dstate s1) (pending s1 p))
    case True
    obtain rs where 8:∀ r ∈ set rs . fst r ≠ p
      and 9:dstate t2 = dstate s1 ⋆ rs
    proof −
      obtain rs where dstate s2 = dstate s1 ⋆ rs
      ∧ (∀ r ∈ set rs . fst r ≠ p) using 1 5 6 True Lin1(1) by force
    hence ∀ r ∈ set rs . fst r ≠ p and dstate t2 = dstate s1 ⋆ rs
      using Lin1(1) by auto
    thus ?thesis using that by blast
  qed
  have 10:dstate t1 ≼ dstate t2 using 4 5 by auto
  obtain rs'' where 11:dstate t2 = dstate t1 ⋆ rs''
    and 12:set rs'' ⊆ set rs
      using consistency 2 9 10 by simp (metis less-eq-def)
  have 13:∀ r ∈ set rs'' . fst r ≠ p using 8 12 by blast
  thus ∃ rs . dstate t2 = dstate t1 ⋆ rs ∧ (∀ r ∈ set rs . fst r ≠ p)
    using 11 13 by auto
next
  case False
  obtain rs'' where 8:dstate t2 = dstate t1 ⋆ rs''
    and 9:∀ r ∈ set rs'' . fst r = p ⟶ r = pending t1 p
  proof −
    obtain rs where 8:∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
      and 9:dstate t2 = dstate s1 ⋆ rs
    proof −
      obtain rs where dstate s2 = dstate s1 ⋆ rs
        ∧ (∀ r ∈ set rs . fst r  = p ⟶ r = pending s1 p)
          using 1 5 6 False Lin1(1) by force
      hence ∀ r ∈ set rs . fst r  = p ⟶ r = pending s1 p
        and dstate t2 = dstate s1 ⋆ rs
          using Lin1(1) by auto
      thus ?thesis using that by blast
    qed
    have 10:dstate t1 ≼ dstate t2 using 4 5 by auto
    obtain rs'' where 11:dstate t2 = dstate t1 ⋆ rs''
      and 12:set rs'' ⊆ set rs
        using consistency 2 9 10 by simp (metis less-eq-def)
    have 13:∀ r ∈ set rs'' . fst r = p ⟶ r = pending s1 p
      using 8 12 by blast
    have dstate t2 = dstate t1 ⋆ rs''
      ∧ (∀ r ∈ set rs'' . fst r = p ⟶ r = pending t1 p)
        using 11 13 Lin1(1) by auto
    thus ?thesis using that by blast
  qed
  have 10:dstate t1 ⋆ rs''
    = dstate t1 ⋆ (filter (λ r . r ≠ pending t1 p) rs'')
    using 7 idem-star by blast
  have 11:∀ r ∈ set (filter (λ r . r ≠ pending t1 p) rs'') .
    fst r ≠ p using 9 by force
  show ∃ rs . dstate t2 = dstate t1 ⋆ rs ∧ (∀ r ∈ set rs . fst r ≠ p)
```

```
        using 8 10 11 by metis
      qed
    qed
  next
    case Lin2
    assume 1:P17 (s1,s2)
    { fix p
      assume 2:status s1 p ≠ Aborted
      have ∃ rs' . dstate t2 = dstate s2 ⋆ rs'
        ∧ (∀ r ∈ set rs' . fst r ≠ p)
      proof −
        obtain rs' where 5:dstate t2 = dstate s2 ⋆ rs'
          and 6:rs' ∈ pendingSeqs s2 using Lin2(1) by force
        have 7:∀ r ∈ set rs' . fst r ≠ p
        proof (rule ballI)
          fix r
          assume r ∈ set rs'
          with 6 have r ∈ pendingReqs s2 by (auto simp add:pendingSeqs-def)
          moreover
          have P2 (s1,s2) using reach assms P2-invariant
            by (metis invariant-def)
          moreover
          have status s2 p = Sleep
          proof −
            have P6 (s1,s2) using reach assms P6-invariant
              by (metis invariant-def)
            thus ?thesis using 2 Lin2(1) by force
          qed
          ultimately show fst r ≠ p by (auto simp add:pendingReqs-def)
        qed
        show ?thesis using 5 7 by force
      qed }
      note a = this
    show P17 (t1,t2)
    proof auto
      fix p
      assume 2:initialized t2 and 3:status t1 p = Ready
      obtain rs where dstate s2 = dstate s1 ⋆ rs
        and ∀ r ∈ set rs . fst r ≠ p
      proof −
        have initialized s2 and status s1 p = Ready
          using Lin2(1) 2 3 by auto
        thus ?thesis using that 1 by fastforce
      qed
      moreover
  obtain rs′ where dstate t2 = dstate s2 ⋆ rs′
    and ∀ r ∈ set rs′ . fst r ≠ p using a 3 Lin2(1)
      by (metis SLin-status.distinct(11))
  ultimately show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r ≠ p) using Lin2(1)
      by auto (metis UnE exec-append set-append)
next
  fix p
  assume 2:initialized t2 and 3:status t1 p = Pending
    and 4:contains (dstate t1) (pending t1 p)
  obtain rs where dstate s2 = dstate s1 ⋆ rs
    and ∀ r ∈ set rs . fst r ≠ p
  proof −
    have initialized s2 and status s1 p = Pending
      ∧ contains (dstate s1) (pending s1 p)
        using Lin2(1) 2 3 4 by auto
    thus ?thesis using that 1 by fastforce
  qed
  moreover
  obtain rs′ where dstate t2 = dstate s2 ⋆ rs′
    and ∀ r ∈ set rs′ . fst r ≠ p using a 3 Lin2(1)
      by (metis SLin-status.distinct(9))
  ultimately show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r ≠ p) using Lin2(1)
      by auto (metis UnE exec-append set-append)
next
  fix p
  assume 2:initialized t2 and 3:status t1 p = Pending
    and 4:¬ contains (dstate t1) (pending t1 p)
  obtain rs where dstate s2 = dstate s1 ⋆ rs
    and ∀ r ∈ set rs . fst r = p ⟶ r = pending s1 p
  proof −
    have initialized s2 and status s1 p = Pending
      ∧ ¬ contains (dstate s1) (pending s1 p)
        using Lin2(1) 2 3 4 by auto
    thus ?thesis using that 1 by fastforce
  qed
  moreover
  obtain rs′ where dstate t2 = dstate s2 ⋆ rs′
    and ∀ r ∈ set rs′ . fst r ≠ p using a 3 Lin2(1)
      by (metis SLin-status.distinct(9))
  ultimately show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending t1 p)
      using Lin2(1)
        by auto (metis UnE exec-append set-append)
  qed
next
  case Reco2
  assume 0:P17 (s1,s2)
  obtain rs' where 1:dstate t2 = dstate t1 ⋆ rs'
    and 2:set rs' ⊆ pendingReqs s1 ∪ pendingReqs s2
  proof -
    obtain ivs rs where 1:ivs ⊆ initVals s2 and 2:ivs ≠ {}
      and 3:dstate t2 = ⨅ivs ⋆ rs and 4:rs ∈ pendingSeqs s2
      using Reco2(1) by (simp add:safeInits-def initSets-def, force)
    obtain rs'' where set rs'' ⊆ pendingReqs s1
      and ⨅ivs = dstate s1 ⋆ rs''
    proof -
      have P8a (s1,s2) using reach assms P8a-invariant
        by (metis invariant-def)
      thus ?thesis using that using 1 2
        by (auto simp add:initSets-def pendingSeqs-def)
    qed
    hence dstate t2 = dstate t1 ⋆ (rs''@rs)
      ∧ set rs'' ⊆ pendingReqs s1
        ∧ set rs ⊆ pendingReqs s2
      using 3 4 Reco2(1) 4
        by (metis exec-append mem-Collect-eq pendingSeqs-def)
    thus ?thesis using that by force
  qed
  { fix p r
    assume 1:r ∈ pendingReqs s2
      and 2:status s1 p ≠ Aborted
    have fst r ≠ p
    proof -
      have P2 (s1,s2) using reach P2-invariant assms
        by (metis invariant-def)
      moreover
      have P6 (s1,s2) using reach P6-invariant assms
        by (metis invariant-def)
      ultimately show ?thesis using 1 2 Reco2(1)
        by (simp add:pendingReqs-def)
          (metis SLin-status.distinct(1,5))
      qed }
  note 3 = this
  { fix r p
    assume 1:r ∈ pendingReqs s1 and 2:fst r = p
      and 3:status s1 p = Pending
    have r = pending s1 p
    proof -
    have P1 (s1,s2) using reach P1-invariant assms
      by (metis invariant-def)
    thus ?thesis using 1 2 3
      by (auto simp add:pendingReqs-def)
  qed }
note 10 = this
show P17 (t1,t2)
proof (auto)
  fix p
  assume 4:status t1 p = Ready
  show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r ≠ p)
  proof -
    { fix r
      assume 5:r ∈ pendingReqs s1
      have fst r ≠ p
      proof -
        have P1 (s1,s2) using reach P1-invariant assms
          by (metis invariant-def)
        with 4 5 Reco2(1) show ?thesis
          by (auto simp add:pendingReqs-def)
      qed }
    moreover
    have ⋀ r . r ∈ pendingReqs s2 ⟹ fst r ≠ p
      using 3 4 Reco2(1) by auto
    ultimately show ?thesis using 1 2 by blast
  qed
next
  fix p
  assume 4:status t1 p = Pending
    and 5:contains (dstate t1) (pending t1 p)
  show ∃ rs . dstate t2 = dstate t1 ⋆ rs
    ∧ (∀ r ∈ set rs . fst r ≠ p)
  proof -
    let ?rs = filter (λ r . r ≠ pending t1 p) rs'
    have dstate t2 = dstate t1 ⋆ ?rs
      using 5 1 idem-star by metis
    moreover
    { fix r
      assume r ∈ set ?rs
      have fst r ≠ p
      proof -
        { fix r
          assume 6:r ∈ set rs' and 7:fst r = p
          have r = pending s1 p
```

```
              proof −
                have ⋀ r . r ∈ pendingReqs s2 ⟹ fst r ≠ p
                  using 3 4 Reco2(1) by auto
                moreover
                have ⋀ r . r ∈ pendingReqs s1 ⟹ fst r = p
                  ⟹ r = pending s1 p
                    using 10 4 Reco2(1) by auto
                ultimately show ?thesis using 2 6 7
                  by (metis (lifting, no-types) UnE subsetD)
              qed }
            thus ?thesis using ‹r ∈ set ?rs› Reco2(1) by fastforce
          qed }
        ultimately show ?thesis by blast
      qed
    next
      fix p
      assume 4:status t1 p = Pending
        and 5:¬ contains (dstate t1) (pending t1 p)
      show ∃ rs . dstate t2 = dstate t1 ⋆ rs
        ∧ (∀ r ∈ set rs . fst r = p ⟶ r = pending t1 p)
      proof −
        have ⋀ r . r ∈ pendingReqs s2 ⟹ fst r ≠ p
          using 3 4 Reco2(1) by auto
        moreover
        have ⋀ r . r ∈ pendingReqs s1 ⟹ fst r = p
          ⟹ r = pending s1 p
            using 10 4 Reco2(1) by auto
        ultimately show ?thesis using 1 2 Reco2(1)
          by (metis (lifting, no-types) UnE set-rev-mp)
      qed
    qed
  qed
qed

lemma P21-invariant:
shows invariant (composition) P21
proof (rule invariantI, simp-all only:split-paired-all)
  fix s1 s2
  assume (s1,s2) ∈ ioa.start (composition)
  thus P21 (s1,s2) using ids by (auto simp add:comp-simps)
next
  fix s1 s2 t1 t2 a
  assume hyp: P21 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
  and reach: reachable (composition) (s1,s2)
  show P21 (t1,t2)
```

```
proof (cases initialized t2)
  case True
  moreover
  have P12:P12 (t1,t2) using P12-invariant reach trans
    by (metis invariant-def reachable-n)
  moreover
  have P11:P11 (t1,t2) using P11-invariant reach trans
    by (metis IOA.invariant-def reachable-n)
  moreover
  have P20:P20 (t1,t2) using P20-invariant reach trans
    by (metis IOA.invariant-def reachable-n)
  ultimately show P21 (t1,t2) by simp
    (metis pre-RDR.trans)
next
  case False
  show P21 (t1,t2) using trans
  proof (cases rule:trans-elim)
    case (Switch2 p c av)
    obtain av where abortVals t2 = abortVals s2 ∪ {av}
      and ⊓(abortVals s1) ⪯ av
    proof −
      obtain ivs rs where 1:abortVals t2 = abortVals s2 ∪ {⊓ivs ⋆ rs}
        and 2:ivs ⊆ initVals s2 and 3:ivs ≠ {}
        using False Switch2(1) by (auto simp add:safeAborts-def
          uninitAborts-def initSets-def)
      have 4:⊓(abortVals s1) ⪯ ⊓ivs
      proof −
        have P11 (s1,s2) using reach assms P11-invariant
          by (metis invariant-def)
        moreover
        have P13 (s1,s2) using reach assms P13-invariant
          by (metis invariant-def)
        ultimately show ?thesis using 2 3 antimono by simp
      qed
      show ?thesis using that 1 4 by simp
        (metis coboundedI2 less-eq-def orderE)
    qed
    with hyp show ?thesis using Switch2(1) by simp
  next
    case (Switch1 p c av)
    show ?thesis
    proof (cases abortVals s1 = {})
      case False
      have ⊓ (abortVals t1) ⪯ ⊓ (abortVals s1)
      proof −
```

```
          obtain av where abortVals t1 = abortVals s1 ∪ {av}
            using Switch1(1) by auto
          moreover
          have P13 (s1,s2) using reach assms P13-invariant
            by (metis invariant-def)
          ultimately show ?thesis using False by simp
        qed
        moreover have abortVals t2 = abortVals s2
          using Switch1(1) by auto
        ultimately show ?thesis using hyp
          by auto (metis coboundedI2 orderE)
      next
        case True
        have abortVals t2 = {}
        proof -
          have P19 (s1,s2) using reach assms P19-invariant
            by (metis invariant-def)
          thus ?thesis using True Switch1(1) by auto
        qed
        thus ?thesis by auto
      qed
    next
      case (Invoke1 p c)
      thus ?thesis using hyp by simp
    next
      case (Invoke2 p c)
      thus ?thesis using hyp by simp
    next
      case (Response1 p ou)
      thus ?thesis using hyp by simp
    next
      case (Response2 p ou)
      thus ?thesis using hyp by simp
    next
      case Lin1
      thus ?thesis using hyp by auto
    next
      case Lin2
      thus ?thesis using hyp by auto
    next
      case Reco2
      thus ?thesis using hyp by auto
    qed
  qed
qed
```

```
lemma P22-invariant:
shows invariant (composition) P22
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  have P9:P9 (s1,s2) using P9-invariant assms 1
    by (simp add:invariant-def)
  show P22 (s1,s2)
  proof (simp only:P22.simps, rule impI)
    assume initialized s2
    show dstate (f (s1,s2)) = dstate s2
    proof (cases dstate s2 = ⊥)
      case False
      thus ?thesis by auto
    next
      case True
      show dstate  (f (s1,s2)) = dstate s2
      proof −
        have dstate s1 ≼ dstate s2
          using ‹initialized s2› and ‹P9 (s1,s2)›
            by auto
        hence dstate s1 = dstate s2 using True
          by (metis antisym bot)
        thus ?thesis by auto
      qed
    qed
  qed
qed

lemma P23-invariant:
shows invariant (composition) P23
proof (auto simp only:invariant-def)
  fix s1 s2
  assume 1:reachable (composition) (s1,s2)
  show P23 (s1,s2)
  proof (simp only:P23.simps, clarify)
    fix rs
    assume 2:¬initialized s2 and 3:rs∈pendingSeqs s1
    show rs∈ pendingSeqs (f (s1,s2))
    proof −
      { fix r
        assume 3:r ∈ pendingReqs s1
        have 4:status s1 (fst r) = Pending ∨ status s1 (fst r) = Aborted
          and 5:pending s1 (fst r) = r
    proof -
      have P1 (s1,s2) using 1 P1-invariant
        by (metis invariant-def)
      thus status s1 (fst r) = Pending ∨ status s1 (fst r) = Aborted
      and pending s1 (fst r) = r
        using 3 by (auto simp add:pendingReqs-def)
    qed
    have r ∈ pendingReqs (f (s1,s2)) using 4
    proof
      assume status s1 (fst r) = Pending
      with 5 show ?thesis by (auto simp add:pendingReqs-def)
        (metis SLin-status.distinct(9))
    next
      assume 6:status s1 (fst r) = Aborted
      have 7:pending s1 (fst r) = pending s2 (fst r)
        ∧ status s2 (fst r) ∈ {Pending,Aborted}
      proof -
        have P7 (s1,s2) using 1 P7-invariant
          by (metis invariant-def)
        thus ?thesis using 2 6 by auto
      qed
      show ?thesis using 6 5 7 by (simp add:pendingReqs-def, metis)
    qed }
   thus ?thesis using 3 by (auto simp only:pendingSeqs-def)
  qed
 qed
qed

lemma P26-invariant:
shows invariant (composition) P26
proof (rule invariantI, simp-all only:split-paired-all)
 fix s1 s2
 assume (s1,s2) ∈ ioa.start (composition)
 thus P26 (s1,s2) using ids by (auto simp add:comp-simps)
next
 fix s1 s2 t1 t2 a
 assume hyp: P26 (s1,s2) and trans:(s1,s2) −a−composition⟶ (t1,t2)
 and reach:reachable composition (s1,s2)
 show P26 (t1,t2) using trans and hyp
 proof (cases rule:trans-elim)
  case Lin2
  hence 1:dstate s2 ≼ dstate t2
   by auto (metis less-eq-def)
  have 2:t2 = s2⦇dstate := dstate t2⦈ and 3:s1 = t1
   using Lin2(1) by auto
  show ?thesis
  proof (simp, clarify)
    fix p
    assume 4:status t1 p = Aborted
      and 5:¬ contains (dstate t2) (pending t1 p)
    have 6:status s1 p = Aborted using 3 4 by auto
    have 7:pending s1 p = pending t1 p using 3 by simp
    have 8:¬ contains (dstate s2) (pending s1 p)
      using 1 5 7
        by simp (metis contains-star less-eq-def)
    have 11:status s2 p ∈ {Pending,Aborted}
      and 9:pending s1 p = pending s2 p using hyp 6 8 by auto
    show (status t2 p = Pending ∨ status t2 p = Aborted)
      ∧ pending t1 p = pending t2 p
    proof −
      from 2 have pending s2 = pending t2
          and status s2 = status t2 by ((cases s2, cases t2, auto)+)
      thus ?thesis using 9 3 11 by auto
    qed
  qed
next
  case Reco2
  show ?thesis
  proof (simp,clarify)
    fix p
    assume 1:status t1 p = Aborted
    have 2:status s1 p = Aborted and 3:¬initialized s2
      using 1 Reco2(1) by auto
    have 4:P7 (s1,s2) using reach P7-invariant
      by (metis invariant-def)
    have 5:status s2 p ∈ {Pending,Aborted}
    and 6:pending s1 p = pending s2 p using 3 4 2 by auto
    show (status t2 p = Pending ∨ status t2 p = Aborted)
      ∧ pending t1 p = pending t2 p using 5 6 Reco2(1) by auto
  qed
next
  case Lin1
  thus ?thesis using hyp by force
next
  case Response1
  thus ?thesis using hyp by force
next
  case Response2
  thus ?thesis using hyp by force
next
```

```
    case Invoke2
    thus ?thesis using hyp by force
  next
    case Switch1
    thus ?thesis using hyp by force
  next
    case Switch2
    thus ?thesis using hyp by force
  next
    case Invoke1
    thus ?thesis using hyp by force
  qed
qed


lemma P25-invariant:
shows invariant (composition) P25
proof (auto simp only:invariant-def)
  fix s1 s2
  assume reach:reachable (composition) (s1,s2)
  show P25 (s1,s2)
  proof (simp only:P25.simps, clarify)
    fix ivs
    assume 1:ivs ∈ initSets s2 and 2:initialized s2
      and 3:dstate s2 ⪯ ⨅ivs
    obtain rs' where 4:dstate s2 ⋆ rs' = ⨅ivs
    and 5:rs' ∈ pendingSeqs s1 and 6:∀ r ∈ set rs' . ¬ contains (dstate s2) r
    proof –
      have 5:dstate s1 ⪯ dstate s2
      proof –
        have P9:P9 (s1,s2) using P9-invariant reach
          by (simp add:invariant-def)
        thus ?thesis using 2 by auto
      qed
      obtain rs where 6:⨅ivs = dstate s1 ⋆ rs and 7:rs ∈ pendingSeqs s1
      proof –
        have P8a:P8a (s1,s2) using P8a-invariant reach
          by (simp add:invariant-def)
        thus ?thesis using that 1 by auto
      qed
      have ∃ rs' . dstate s2 ⋆ rs' = ⨅ ivs ∧ rs' ∈ pendingSeqs s1
        using 3 5 6 7 consistency[of dstate s1 dstate s2 ⨅ivs rs]
          by (force simp add:pendingSeqs-def)
      with this obtain rs' where ⨅ivs = dstate s2 ⋆ rs'
        and rs' ∈ pendingSeqs s1 by metis
      with this show ?thesis using idem-star2 that
        by (metis mem-Collect-eq pendingSeqs-def subset-trans)
    qed
    have 7:rs' ∈ pendingSeqs (f (s1,s2))
    proof −
      { fix r
        assume r ∈ set rs'
        with this obtain p where 8:status s1 p = Pending
          ∨ status s1 p = Aborted
        and 9:r = pending s1 p
          using 5 by (auto simp add:pendingReqs-def pendingSeqs-def)
        from 8 have r ∈ pendingReqs (f (s1,s2))
        proof
          assume status s1 p = Pending
          thus ?thesis using 9 by (simp add:pendingReqs-def)
            (metis SLin-status.distinct(9))
        next
          assume 10:status s1 p = Aborted
          hence status (f (s1,s2)) p = status s2 p
            and pending (f (s1,s2)) p = pending s2 p by simp-all
          moreover
          have status s2 p ∈ {Pending,Aborted} ∧ pending s2 p = pending s1 p
          proof −
            have ¬ contains (dstate s2) r
              using 6 ‹r ∈ set rs'› by simp
            moreover
            have P26 (s1,s2) using reach P26-invariant
              by (metis invariant-def)
            ultimately show ?thesis using 10 9 by force
          qed
         ultimately show ?thesis using 9 by (simp only:pendingReqs-def, force)
        qed }
      thus ?thesis by (auto simp add:pendingSeqs-def)
    qed
    show ∃ rs ∈ pendingSeqs (f (s1,s2)) . ⨅ivs = dstate s2 ⋆ rs
      using 4 7 by force
  qed
qed
```

## 7.4 Proof of the Idempotence Theorem

```
theorem idempotence:
  shows ((composition) =<| (ioa 0 id2))
```

```
proof −
  have same-input-sig:inp (composition) = inp (ioa 0 id2)
    — First we show that both automata have the same input and output signature

      using  ids by auto
  moreover
  have same-output-sig:out (composition) = out (ioa 0 id2)
    — Then we show that output signatures match
    using ids by auto
  moreover
  have traces (composition) ⊆ traces (ioa 0 id2)
    — Finally we show trace inclusion
  proof −
    have ext (composition) = ext (ioa 0 id2)
      — First we show that they have the same external signature
      using same-input-sig and same-output-sig by simp
    moreover
    have is-ref-map f (composition) (ioa 0 id2)
      — Then we show that f-comp is a refinement mapping
    proof (auto simp only:is-ref-map-def)
      fix s1 s2
      assume 1:(s1,s2) ∈ ioa.start (composition)
      show f (s1,s2) ∈ ioa.start (ioa 0 id2)
      proof −
        have 2:ioa.start (ioa 0 id2) = start (0::nat) by simp
        have 3:ioa.start (composition)
          = start (0::nat) × start id1 by fastforce
        show ?thesis
          using assms 1 2 3 by simp
      qed
    next
      fix s1 s2 t1 t2 :: ('a,'b,'c)SLin-state and a :: ('a,'b,'c,'d)SLin-action
      assume reach:reachable (composition) (s1,s2)
      and trans:(s1,s2) −a−(composition)−→ (t1,t2)
      def u ≡ f (s1,s2) def u' ≡ f (t1,t2)
```

Lemmas and invariants

```
      have pendingReqs s2 ⊆ pendingReqs u
      proof −
        have P6 (s1,s2) using reach assms P6-invariant
          by (metis invariant-def)
        thus ?thesis
          by (force simp add:pendingReqs-def u-def)
      qed
      note lem1 = this
```

```
have initialized u by (auto simp add:u-def)
have P1 (s1,s2) and P1 (t1,t2) using reach assms P1-invariant
  trans invariant-def by (metis , metis reachable-n)
have P6 (s1,s2) and P6 (t1,t2) using reach assms P6-invariant
  trans invariant-def by (metis , metis reachable-n)
have P7 (s1,s2) and P7 (t1,t2) using reach assms P7-invariant
  trans invariant-def by (metis , metis reachable-n)
have P8 (s1,s2) and P8 (t1,t2) using reach assms P8-invariant
  trans invariant-def by (metis , metis reachable-n)
have P9 (s1,s2) and P9 (t1,t2) using reach assms P9-invariant
  trans invariant-def by (metis , metis reachable-n)
have P10 (s1,s2) and P10 (t1,t2) using reach assms P10-invariant
  trans invariant-def by (metis , metis reachable-n)
have P13 (s1,s2) and P13 (t1,t2) using reach assms P13-invariant
  trans invariant-def by (metis , metis reachable-n)
have P15 (s1,s2) and P15 (t1,t2) using reach assms P15-invariant
  trans invariant-def by (metis , metis reachable-n)
have P16 (s1,s2) and P16 (t1,t2) using reach assms P16-invariant
  trans invariant-def by (metis , metis reachable-n)
have P17 (s1,s2) and P17 (t1,t2) using reach assms P17-invariant
  trans invariant-def by (metis , metis reachable-n)
have P19 (s1,s2) and P19 (t1,t2) using reach assms P19-invariant
  trans invariant-def by (metis , metis reachable-n)
have P21 (s1,s2) and P21 (t1,t2) using reach assms P21-invariant
  trans invariant-def by (metis , metis reachable-n)
have P22 (s1,s2) and P22 (t1,t2) using reach assms P22-invariant
  trans invariant-def by (metis , metis reachable-n)
have P25 (s1,s2) and P25 (t1,t2) using reach assms P25-invariant
  trans invariant-def by (metis , metis reachable-n)
have P8a (s1,s2) and P8a (t1,t2) using reach assms P8a-invariant
  trans invariant-def by (metis , metis reachable-n)
have P23 (s1,s2) and P23 (t1,t2) using reach assms P23-invariant
  trans invariant-def by (metis , metis reachable-n)

show ∃ e . refines e (s1,s2) a (t1,t2) (ioa 0 id2) f
  using assms and trans
proof (cases rule:trans-elim)
  case (Invoke1 i p c)
  let ?e = (u,[(a,u′)])
  have 1:is-exec-frag-of (ioa 0 id2) ?e
  proof −
    have 1:status s1 p = Ready and 2:t2 = s2
    and 3:t1 = s1⦇pending := (pending s1)(p := (p,c)),
      status := (status s1)(p := Pending)⦈
      using Invoke1(1) by auto
```

```
          have 4:status u p = Ready using 1 u-def by auto
          have 5:u' = u⦇pending := (pending u)(p := (p,c)),
            status := (status u)(p := Pending)⦈
              using 2 3 u-def u'-def by auto
          have 6:Inv p c u u' using 4 5 by force
          show ?thesis using 6 Invoke1(3) ids by simp
        qed
        have 2:a ∈ ext (ioa 0 id2) and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
       using Invoke1(2,3) ids by (auto simp add:trace-def schedule-def filter-act-def)
        show ?thesis using 1 2 3
          by (simp only:refines-def u-def u'-def)
            (metis fst-conv last-state.simps(2) snd-conv)
      next

        case (Invoke2 i p c)
        let ?e = (u,[(a,u')])
        have 1:is-exec-frag-of (ioa 0 id2) ?e
        proof -
          have 1:status s2 p = Ready and 2:t1 = s1
          and 3:t2 = s2⦇pending := (pending s2)(p := (p,c)),
            status := (status s2)(p := Pending)⦈
            using Invoke2(1) by auto
          have 4:status u p = Ready using 1 u-def ‹P6 (s1,s2)› by auto
          have 5:u' = u⦇pending := (pending u)(p := (p,c)),
            status := (status u)(p := Pending)⦈
              using 2 3 u-def u'-def ‹P6 (t1,t2)› by fastforce
          have 6:Inv p c u u' using 4 5 by force
          show ?thesis using 6 Invoke2(3) ids by simp
        qed
        have 2:a ∈ ext (ioa 0 id2)
        and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
              using Invoke2(2,3) assms by (auto simp add:trace-def schedule-def
filter-act-def)
        show ?thesis using 1 2 3
          by (simp only:refines-def u-def u'-def)
            (metis fst-conv last-state.simps(2) snd-conv)
      next

        case (Response2 i p ou)
        let ?e = (u,[(a,u')])
        have 1:is-exec-frag-of (ioa 0 id2) ?e
        proof -
          have 1:status s1 p = Aborted ∧ status t1 p = Aborted
          proof -
            show ?thesis using ‹P6 (s1,s2)› ‹P6 (t1,t2)›
        Response2(1) by force
    qed
    have 2:status u p = Pending ∧ initialized u
      using 1 Response2(1) u-def by auto
    have  3:u′ = u⦇status := (status u)(p := Ready)⦈
      using 1 Response2(1) u-def u′-def
        by (cases u, cases u′, auto)
    have 4:ou = γ (dstate u) (pending u p) ∧ contains (dstate u) (pending u p)
    proof (cases dstate s2 = ⊥)
      case False
      thus ?thesis using 1 Response2(1) u-def by auto
    next
      case True
      have dstate s1 = dstate s2
      proof −
        have dstate s1 ≼ dstate s2
          using Response2(1) ‹P9 (s1,s2)› by auto
        with True show ?thesis by (metis antisym bot)
      qed
      thus ?thesis using 1 Response2(1) u-def by auto
    qed
    show ?thesis using 2 3 4 Response2(3) ids by auto
  qed
  have 2:a ∈ ext (ioa 0 id2)
  and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
    using Response2(2,3) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  show ?thesis using 1 2 3
    by (simp only:refines-def u-def u′-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next

  case (Response1 i p ou)
  let ?e = (u,[(a,u′)])
  have 1:is-exec-frag-of (ioa 0 id2) ?e
  proof (cases dstate s2 = ⊥)
    case True
    have 1:status u p = Pending ∧ initialized u
      using Response1(1) u-def by auto
    have  2:u′ = u⦇status := (status u)(p := Ready)⦈
      using Response1(1) u-def u′-def
        by (cases u, cases u′, auto)
    have 3:ou = γ (dstate u) (pending u p)
      ∧ contains (dstate u) (pending u p)
    using Response1(1) True u-def by auto
  show ?thesis using 1 2 3 ‹initialized u› Response1(3) ids by auto
next
  case False
  have 1:status u p = Pending ∧ initialized u
    using Response1(1) u-def by auto
  have  2:u′ = u⦇status := (status u)(p := Ready)⦈
    using Response1(1) u-def u′-def
      by (cases u, cases u′, auto)
  have 3:ou = γ (dstate u) (pending u p)
    and 4:contains (dstate u) (pending u p)
  proof −
    have 2:contains (dstate s1) (pending s1 p)
      using Response1(1) by auto
    show contains (dstate u) (pending u p)
    proof −
      have 3:dstate s1 ⪯ dstate u
      proof −
        have initialized s2 using ‹P16 (s1,s2)› False
          by auto
        thus ?thesis using ‹P9 (s1,s2)› u-def False refl by simp
      qed
      have 4:pending s1 p = pending u p
        using u-def Response1(1) by force
      show ?thesis
        using 2 3 4 by (metis contains-star less-eq-def)
    qed
    have 4:γ (dstate s1) (pending s1 p) = γ (dstate u) (pending u p)
    proof −
      have 4:pending s1 p = pending u p
        using u-def Response1(1) by force
      obtain rs where 5:dstate u = dstate s1 ⋆ rs
        and 6:∀ r ∈ set rs . fst r ≠ p
      proof −
        have 7:dstate u = dstate s2 using u-def False by simp
        have 6:status s1 p = Pending
         ∧ contains (dstate s1) (pending s1 p)
            using Response1(1) by force
        have 8:initialized s2 using False ‹P16 (s1,s2)›
          by auto
        show ?thesis using that ‹P17 (s1,s2)› 6 8 7 by fastforce
      qed
      have 7:fst (pending s1 p) = p
        using Response1(1) ‹P1 (s1,s2)› by auto
      show ?thesis using 4 5 6 7 2 idem2-star by auto
      qed
      thus ou = γ (dstate u) (pending u p)
        using Response1(1) by simp
    qed
    thus ?thesis using 1 2 3 Response1(3) ids by auto
  qed
  have 2:a ∈ ext (ioa 0 id2)
  and 3:trace (ioa.asig (ioa 0 id2)) ?e = [a]
    using Response1(2,3) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  show ?thesis using 1 2 3
    by (simp only:refines-def u-def u'-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next

  case (Reco2)
  let ?e = (u,[(Linearize 0,u')])
  have is-exec-frag-of (ioa 0 id2) ?e
  proof -
    obtain rs where 1:rs ∈ pendingSeqs u
      and 2:dstate u' = dstate u ⋆ rs
      and 3:∀ av ∈ abortVals u . dstate u' ≼ av
    proof -
      obtain rs where set rs ⊆ pendingReqs s1 ∪ pendingReqs s2
      and dstate t2 = dstate s1 ⋆ rs
      and ∀ av ∈ abortVals s2 . dstate t2 ≼ av
      proof -
        obtain ivs rs where 3:ivs ⊆ initVals s2 and 4:ivs ≠ {}
        and 5:dstate t2 = ⨅ivs ⋆ rs and 7:rs ∈ pendingSeqs s2
        and 6:∀ av ∈ abortVals s2 . dstate t2 ≼ av
          using Reco2(1)
          by (auto simp add:safeInits-def)
            (metis (lifting, mono-tags) initSets-def mem-Collect-eq)
        obtain rs' where ⨅ivs = dstate s1 ⋆ rs'
        and set rs' ⊆ pendingReqs s1
        proof -
          { fix iv
            assume 7:iv ∈ ivs
            have ∃ rs . set rs ⊆ pendingReqs s1
              ∧ iv = dstate s1 ⋆ rs
                using ‹P8 (s1,s2)› 7 3 by auto
                  (metis mem-Collect-eq pendingSeqs-def set-rev-mp) }
          moreover have finite ivs using ‹P13 (s1,s2)› 3
              by (metis P13.simps rev-finite-subset)
          ultimately show ?thesis using that glb-common-set 4
```

```
            by metis
        qed
        hence dstate t2 = dstate s1 ⋆ (rs'@rs)
          ∧ set (rs'@rs) ⊆ pendingReqs s1 ∪ pendingReqs s2 using 5 7
            by (metis (lifting, no-types) Un-commute Un-mono
                exec-append mem-Collect-eq pendingSeqs-def set-append)
        thus ?thesis using that 6 by blast
      qed
      moreover
      have pendingReqs s1 ∪ pendingReqs s2 ⊆ pendingReqs u
      proof –
        note ‹pendingReqs s2 ⊆ pendingReqs u›
        moreover
        have pendingReqs s1 ⊆ pendingReqs u
          using Reco2(1) ‹P7 (s1,s2)›
            by (auto simp add:pendingReqs-def u-def)
        ultimately show ?thesis by auto
      qed
      moreover
      have abortVals u = abortVals s2 by (auto simp add:u-def)
      moreover
      have dstate u = dstate s1 using ‹P16 (s1,s2)›
        Reco2(1) u-def by force
      moreover
      have dstate u' = dstate t2
        using Reco2(1) ‹P22 (t1,t2)› by (auto simp add:u'-def)
      ultimately show ?thesis using that
        by (auto simp add:pendingSeqs-def, blast)
    qed
    moreover
    have u' = u(|dstate := dstate u ⋆ rs|)
      using 2 Reco2(1) u-def u'-def by force
    moreover
    note ‹initialized u›
    ultimately show ?thesis by auto
  qed
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Reco2(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  ultimately show ?thesis
    by (simp only:refines-def u-def u'-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next
```

```
  case (Switch1 p c av)
  let ?e = (u,[])
  have is-exec-frag-of (ioa 0 id2) ?e by auto
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Switch1(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  moreover
  have u = u' using Switch1(1) u-def u'-def by auto
  ultimately show ?thesis
    using refines-def[of ?e (s1,s2) a (t1,t2) ioa 0 id2 f]
      u-def u'-def by (metis last-state.simps(1) fst-conv)
next

  case Lin2
  let ?e = (u,[(Linearize 0,u')])
  have is-exec-frag-of (ioa 0 id2) ?e
  proof −
    have u' = u⦇dstate := dstate u'⦈ using Lin2(1)
      by (auto simp add:u-def u'-def)
    moreover
    note ‹initialized u›
    moreover
    obtain rs where dstate u' = dstate u ⋆ rs
      and rs ∈ pendingSeqs u
      and ∀ av ∈ abortVals u . dstate u' ≼ av
    proof −
      obtain rs where 1:dstate t2 = dstate s2 ⋆ rs
        and 2:rs ∈ pendingSeqs s2
        and 3:∀ av ∈ abortVals s2 . dstate t2 ≼ av
        using Lin2(1) by force
      have 4:rs ∈ pendingSeqs u
        using  2 and ‹pendingReqs s2 ⊆ pendingReqs u›
          by (metis mem-Collect-eq pendingSeqs-def subset-trans)
      have 5:dstate u' = dstate u ⋆ rs
        and 6:∀ av ∈ abortVals u . dstate u' ≼ av
      proof −
        have 7:dstate u = dstate s2 ∧ dstate u' = dstate t2
          using ‹P22 (s1,s2)› and ‹P22 (t1,t2)› Lin2(1)
            by (auto simp add:u-def u'-def)
        show dstate u' = dstate u ⋆ rs using 7 1 by auto
        show ∀ av ∈ abortVals u . dstate u' ≼ av
        proof −
```

```
          have abortVals s2 = abortVals u by (auto simp add:u-def)
          thus ?thesis using 7 3 by simp
        qed
      qed
      show ?thesis using that 4 5 6 by auto
    qed
    ultimately show ?thesis by auto
  qed
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Lin2(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  ultimately show ?thesis
    by (simp only:refines-def u-def u′-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next

  case Lin1
  have u′ = u⦇dstate := dstate u′⦈ using Lin1(1)
    by (auto simp add:u-def u′-def)
  show ?thesis
  proof (cases initialized s2)
    case False
    let ?e = (u,[(Linearize 0,u′)])
    have is-exec-frag-of (ioa 0 id2) ?e
    proof −
      note ‹u′ = u⦇dstate := dstate u′⦈›
      moreover
      note ‹initialized u›
      moreover
      obtain rs where dstate u′ = dstate u ⋆ rs
        and rs ∈ pendingSeqs u
        and ∀ av ∈ abortVals u . dstate u′ ⪯ av
      proof −
        obtain rs where 1:dstate t1 = dstate s1 ⋆ rs
          and 2:rs ∈ pendingSeqs s1
          and 3:∀ av ∈ abortVals s1 . dstate t1 ⪯ av
          using Lin1(1) by force
        have 5:pendingSeqs s1 ⊆ pendingSeqs u
          using False ‹P7 (s1,s2)›
            by (auto simp add:pendingReqs-def pendingSeqs-def u-def)
        have 6:dstate u = dstate s1 ∧ dstate u′ = dstate t1
          using ‹P16 (s1,s2)› False Lin1(1)
            by (auto simp add:u-def u′-def)
      have 4:∀ av ∈ abortVals u . dstate u′ ⪯ av
      proof (cases abortVals u = {})
        case True
        thus ?thesis by auto
      next
        case False
        have dstate u′ = dstate t1 using 6 by auto
        moreover have abortVals u = abortVals t2
          using Lin1(1) by (auto simp add:u-def)
        moreover have dstate t1 ⪯ ⨅(abortVals t1)
        proof −
          have abortVals t1 = abortVals s1 using Lin1(1) by auto
          moreover have abortVals t1 ≠ {} using False ‹P19 (t1,t2)›
            Lin1(1) by (simp add: u-def)
          ultimately show ?thesis using 3 ‹P13 (t1,t2)›
            by simp (metis boundedI)
        qed
        ultimately show ?thesis using ‹P21 (t1,t2)› 3
          by (metis P21.simps coboundedI2 orderE)
      qed
      show ?thesis using 1 2 3 4 5 6 that by auto
    qed
    ultimately show ?thesis by auto
  qed
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Lin1(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  ultimately show ?thesis
    by (simp only:refines-def u-def u′-def)
      (metis fst-conv last-state.simps(2) snd-conv)
next
  case True
  let ?e = (u,[])
  have is-exec-frag-of (ioa 0 id2) ?e by auto
  moreover
  have a ∉ ext (ioa 0 id2)
  and trace (ioa.asig (ioa 0 id2)) ?e = []
    using Lin1(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
  moreover have last-state ?e = u′
  proof −
    have dstate u = dstate s2 ∧ dstate u′ = dstate t2
      using ‹P22 (s1,s2)› and ‹P22 (t1,t2)› and True and Lin1(1)
```

```
          by (auto simp add:u-def u'-def)
      thus ?thesis using Lin1(1) ⟨u' = u⦇dstate := dstate u'⦈⟩
        by simp
    qed
    ultimately show ?thesis
      using refines-def[of ?e (s1,s2) a (t1,t2) ioa 0 id2 f]
        by (simp only:u-def u'-def, auto)
  qed
next

  case (Switch2 p c av)
  let ?e = (u,[(a,u')])
  have 1:is-exec-frag-of (ioa 0 id2) ?e
  proof −
    have 1:u' = u⦇abortVals := (abortVals u) ∪ {av},
      status := (status u)(p := Aborted)⦈
      and 2:av ∈ safeAborts s2 and 3:status u p = Pending
      and 4:pending u p = (p,c)
    proof −
      have 1:t2 = s2⦇abortVals := (abortVals s2) ∪ {av},
        status := (status s2)(p := Aborted)⦈
        and 2:av ∈ safeAborts s2 and 3:s1 = t1
        and 4:status s2 p = Pending
          using Switch2(1) by auto
      show 5:status u p = Pending using ⟨P6 (s1,s2)⟩ 4
        by (auto simp add:u-def)
      have 6:status u' p = Aborted using ⟨P6 (t1,t2)⟩ 1
        by (auto simp add:u'-def)
      show pending u p = (p,c) using ⟨P6 (s1,s2)⟩ 4 Switch2(1)
        by (auto simp add:u-def)
      show u' = u⦇abortVals := (abortVals u) ∪ {av},
        status := (status u)(p := Aborted)⦈ using 1 3 5 6
          by (auto simp add:u-def u'-def)
      show av ∈ safeAborts s2 using 2 by assumption
    qed
    have 5:av ∈ safeAborts u
    proof (cases initialized s2)
      case True
      hence 6:dstate u = dstate s2 using ⟨P22 (s1,s2)⟩
        by (auto simp add:u-def)
      have (∃ rs ∈ pendingSeqs s2 . av = dstate s2 ⋆ rs)
        ∨ (dstate s2 ≼ av ∧ (∃ ivs ∈ initSets s2 .
          dstate s2 ≼ ⨅ ivs ∧ (∃ rs ∈ pendingSeqs s2 . av = ⨅ ivs ⋆ rs)))
      proof −
        have av ∈ initAborts s2
      using 2 and True by (auto simp add:safeAborts-def)
    thus ?thesis by (auto simp add:initAborts-def)
  qed
  thus ?thesis
  proof
    assume ∃ rs ∈ pendingSeqs s2 . av = dstate s2 ⋆ rs
    moreover note ‹initialized u›
    ultimately show ?thesis using ‹pendingReqs s2 ⊆ pendingReqs u› 6
      by (simp add:safeAborts-def initAborts-def)
        (metis less-eq-def mem-Collect-eq pendingSeqs-def
          sup.coboundedI2 sup.orderE)
  next
    assume 7:dstate s2 ⪯ av ∧ (∃ ivs ∈ initSets s2 .
      dstate s2 ⪯ ⨅ivs ∧ (∃ rs ∈ pendingSeqs s2 . av = ⨅ivs ⋆ rs))
    show ?thesis
    proof −
      have 8:dstate u ⪯ av using 7 6 by auto
      obtain ivs rs' where 9:ivs ∈ initSets s2
        and 10:dstate s2 ⪯ ⨅ivs
        and 11:rs' ∈ pendingSeqs s2 ∧ av = ⨅ivs ⋆ rs'
          using 7 by auto
      have 12:dstate u = dstate s2 using True ‹P22 (s1,s2)›
        by (auto simp add:u-def)
      moreover
      obtain rs where rs ∈ pendingSeqs u and ⨅ivs = dstate s2 ⋆ rs
        using ‹P25 (s1,s2)› True 9 10 by (auto simp add:u-def)
      ultimately have  av = dstate u ⋆ (rs@rs')
        and rs@rs' ∈ pendingSeqs u
        using 11 by (simp-all add:pendingSeqs-def)
          (metis exec-append, metis lem1 subset-trans)
      thus ?thesis using 8 ‹initialized u›
        by (auto simp add:safeAborts-def initAborts-def)
    qed
  qed
next
  case False
  with 2 have 0:av ∈ uninitAborts s2 by (auto simp add:safeAborts-def)
  show ?thesis
  proof −
    obtain ivs rs where 1:ivs ∈ initSets s2
      and 2:rs ∈pendingSeqs s2
      and 3:av = ⨅ivs ⋆ rs
      using 0 by (auto simp add:uninitAborts-def)
    have 4:rs ∈ pendingSeqs u using lem1 2
      by (auto simp add:pendingSeqs-def)
```

```
       have 5:dstate u = dstate s1 using False ‹P10 (s1,s2)›
        by (auto simp add:u-def)
       obtain rs′ where 6:⨅ ivs = dstate s1 ⋆ rs′
        and 7:rs′ ∈ pendingSeqs s1
         using 1 ‹P8a (s1,s2)› by auto
       have 8:rs′ ∈ pendingSeqs u using False ‹P23 (s1,s2)› 7
        by (auto simp add:u-def)
       have 9:av = dstate u ⋆ (rs′@rs) using 3 5 6
        by (metis exec-append)
       have 10:rs′@rs ∈ pendingSeqs u
        using 4 8 by (auto simp add:pendingSeqs-def)
       show ?thesis using 9 10 ‹initialized u›
        by (auto simp add:safeAborts-def initAborts-def less-eq-def)
      qed
     qed
     show ?thesis using 1 3 4 5 Switch2(2) by auto
    qed
    moreover
    have a ∈ ext (ioa 0 id2)
    and trace (ioa.asig (ioa 0 id2)) ?e = [a]
     using Switch2(2) ids
      by (auto simp add:trace-def schedule-def filter-act-def)
    ultimately show ?thesis
     by (simp only:refines-def u-def u′-def)
      (metis fst-conv last-state.simps(2) snd-conv)
   qed
  qed
  ultimately show ?thesis using ref-map-soundness by blast
 qed
 ultimately show ?thesis by (metis ioa-implements-def)
qed

end

end
```

Croix Rouges 12
1007 Lausanne, Switzerland

+41 78 669 64 32
giuliano.losa@epfl.ch

# Giuliano Losa

## Education

*08/2009-present*
EPFL, Switzerland, PhD student in Computer Science, 5th year.

Supervised by Rachid Guerraoui and Viktor Kuncak.

Thesis title: Modularity in the Design of Robust Distributed Algorithms.

*2007-2009* EPFL, Switzerland, Master in Computer Science.

*2005-2009* Supélec, France, Master in Electrical Engineering.

*2003* Baccalauréat Scientifique, option mathématiques, mention très bien.

## Work Experience

*09/2008-03/2009*
IBM T.J. Watson Research Center, USA, Data-Intensive Systems and Analytics Group. Master thesis.

Design and specification of the SPL programming language.

Design and implementation of a distributed object store, using C++.

*07/2007-08/2007*
C.E.A., France, Study and port of a hard real-time operating system on Linux.

## Publications

Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. "Speculative linearizability". In: *PLDI*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 55–66. DOI: `10.1145/2254064.2254072`.

Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. "Abortable Linearizable Modules". In: *The Archive of Formal Proofs*. Ed. by Gerwin Klein, Tobias Nipkow, and Lawrence Paulson. Formal proof development. `http://afp.sf.net/entries/Abortable_Linearizable_Modules.shtml`, 2012.

Dan Alistarh et al. “On the cost of composing shared-memory algorithms”. In: *SPAA*. Ed. by Guy E. Blelloch and Maurice Herlihy. ACM, 2012, pp. 298–307. DOI: `10.1145/2312005.2312057`

Giuliano Losa et al. “CAPSULE: language and system support for efficient state sharing in distributed stream processing systems”. In: *DEBS*. Ed. by François Bry et al. ACM, 2012, pp. 268–277. DOI: `10.1145/2335484.2335514`

Martin Hirzel et al. *SPL Stream Processing Language Specification, IBM Research report RC24897*. Tech. rep. IBM, 2009

### Languages

French: native speaker.

English: excellent.

Italian: fluent.

German: basic.

### Extra-curricular activities

President of “Supélec Rézo”, the student association in charge of the computer network of Supélec’s campus in Gif-Sur-Yvette.