Modularity in the Design of Robust Distributed Algorithms

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the service academique.
Monday 2nd December, 2013, 15:29

Contents

1 Introduction

Complex systems on which we depend on almost every day, like cars, airplanes, the electric grid, or the internet, may contain dozens, hundreds, thousands, or even millions of computers. To deliver their services, these computers often 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 of the system 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 a downloading files at any time, etc. Yet packets must be routed reliably at all times.

A distributed algorithm is *correct* when it never does anything wrong and it eventually delivers it 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 and when a call never reaches the wrong number. 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.

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 quickly the changes dictated by the scheduling policy 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 [92, 87, 19, 104] but, with the exception of the Abstract

framework [41], 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) [63, 97]. SMR is a general technique used to build robust linearizable implementations of data types. SMR algorithms like Paxos [54] or PBFT [15], which are not adaptive, are notoriously hard to understand. The formal correctness proof of Disk Paxos [50], a crash-fault version of Paxos, is about 7000 lines long. Only an informal proof, 35 pages long, of a simplified version of PBFT is known to the authors [14].

Adaptive SMR protocols are even harder. For instance, the Zyzzyva [53] 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 [98]. 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 anew. 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.

The thesis makes the following contributions:

1. We formalize the problem of devising robust adaptive algorithms:

   (a) We propose a formal model of adaptive algorithms. The model abstracts over the scheduling policy, clarifying the task of the switching mechanism.

   (b) We show that the problem of devising robust adaptive algorithms can be simplified by finding *modular properties*, a new notion that we define. A modular property is a correctness property which applies to each strategy of an adaptive distributed algorithm independently and guarantees the correctness of the entire distributed algorithm.

2. We propose a concrete solution, the Speculative Linearizability modular property, to the problem of devising robust adaptive algorithms.

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.

4. We provide supporting material for others to use Speculative Linearizability to design new adaptive algorithms. The supporting material, consisting of TLA+ [62] specifications and Isabelle/HOL [84] theories, allows one to readily use the TLC model checker to debug new adaptive algorithms and to use Isabelle/HOL to prove new adaptive-algorithms correct in a mechanical way.

Speculative Linearizability is a correctness properties simplifying the analysis of *speculative algorithms*. A speculative algorithm is an optimistic adaptive algorithm: A mode behaves as if a particular assumption about the environment holds, achieving high performance if the assumption is true, but can fail 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 user of the system. 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 [53] or adaptive mutual exclusion [51]. 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, incorporating a switching mechanism in the aborting and initializing mode is very challenging. This is the problem that Speculative Linearizability addresses.

Speculative Linearizability builds on the notion of Linearizability [43, 60, 61, 32], 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, to build a *robust* linearizable object, we must build adaptive and extensible linearizable objects, and face the problem of the explosion of the number of cases to reason about.

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 protocol mode can resume 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. Thanks to speculative linearizability, we obtain a speculatively-linearizable adaptive algorithm, QZ, which has the same progress guarantees as Generalized Paxos [56], 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 module to boost its performance for new conditions. Like Generalized Paxos, our algorithm can execute commuting requests in one message round-trip, a practical and common case.

Finally, 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 check whether an algorithm has its intended properties and that can check our proofs. Therefore, we have formalized part of our work in TLA+ and 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. Although all of the algorithms presented in the

thesis have been model checked for small system sizes, only a restricted variant of Speculative Linearizability has been proved correct in Isabelle/HOL [38]. However, mechanical proofs of distributed systems are still a challenge for state of the art verification technology, even in the case of non-adaptive algorithms [54]. The TLA+ specification are one of contributions of the thesis, as they can be used by others to model-check new adaptive algorithms by refinement of Speculative Linearizability.

Our goal of producing mechanically checked proofs led us to use both TLA+ and I/O automata to obtain all the needed features for a formal development: fast prototyping and debugging with TLA+ and accessible formalized meta theory of I/O automata. Our experience with both tools is discussed in chapter 2.

The work presented in this thesis has not been published before. However, preliminary work led to 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 the algorithms 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.

In this chapter we describe the languages and tools that we have used to formalize our work with the goal of proving our main results mechanically in the Isabelle/HOL [84] interactive theorem prover. In the rest of the thesis, we use the theory of I/O automata [69] for informal discussions and the TLA+ [62] language for formal specifications. This chapter is an introduction to the basic concepts of TLA+ and to the theory of I/O automata.

Distributed algorithms can be concisely represented as the composition of several I/O automaton 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 assertions with the TLC model checker [105] integrated in the TLA Toolbox. The TLA Toolbox is a user-friendly Integrated Development Environment for TLA+ specifications. The TLA Toolbox provides a

graphical interface to edit, check, and prove specifications correct. 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. We have used the TLA Toolbox and TLC extensively to check that our specifications are correct. All our TLA+ specification can be found in appendix A. TLA+ specifications can be proved correct in the TLA Toolbox with TLAPS [23]. However TLAPS is still in development at the time of writing and we have preferred using Isabelle/HOL.

Isabelle/HOL 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 [89]. Isabelle/HOL allows writing and interactively proving statements in higher order logic. All proofs are checked using 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 [99], recursive function definitions, modular organisation of specifications with locales [52], etc. The Isar proof language [102] 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 [88], and Sledgehammer [9], which can call external automatic provers and SMT solvers [9] and reconstruct their proofs in Isabelle/HOL. Moreover, the Nitpick tool [10] can search for counterexamples to putative theorems.

We have specified some of our results in Isabelle/HOL and we have attempted mechanical proofs of our main theorems in Isabelle/HOL. Unfortunately our Isabelle/HOL proofs are not yet complete. We relate our experience using Isabelle/HOL in a separate chapter chapter 8.

There are many other specification frameworks targeting the description of distributed systems and their properties. Let us loosely define a specification framework as a collection consisting of mathematical concepts representing aspects of distributed systems, of a formal language in which to specify systems, and of software tools that help write, debug, or prove the properties of a specification. 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) [7], the I/O-automata framework [49], TLA+ [62] (the Temporal Logic of Actions), Reactive Modules [4], Promela and the SPIN model checker [46], the NuSMV mode-checker [20], Bigraphical Reactive Systems [78], Abstract State Machines [11], and process calculi like CSP [45], the π-calculus [79, 80], and Petri nets [91].

In the rest of this chapter we present the theory of I/O automata, restricted to finite traces, TLA+, and we 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 used 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 examples $\langle e_1, \ldots, e_n \rangle$.

If $es = \langle e_1, \ldots, e_n \rangle$ is a sequence and $i \in 1..n$, we write $es\,[i]$ for e_i. 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 noted $Append\,(es, e)$. The set of all sequences of elements of a set E is noted $Seq\,(E)$.

Arrays are multi-dimensional sequences. The elements at positoin i, j of a two-dimensional array A is noted $A\,[i, j]$. Functions F are the more general case of sequences and arrays, associating elements of their domain, $Dom\,(F)$, set to elements of their image set, $Image\,(F)$.

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 [69] to model asynchronous distributed systems. The theory of I/O automata is also described in details in chapter 8 of Lynch's book [67] , 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-enable whereas, to simplify specifications, ours do not.

An I/O automaton can be though of as a *state-machine* plus an *interface*. First, and 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 processes, or 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 processes 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 *refinement mappings* and *history variables*, which are instances of the more general class of *simulation proofs*.

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 by Lynch and Vaandrager in [70].

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

Finally, it is worth noting that 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 developed by Müller and Nipkow [83, 82], parts of which are still maintained in the Archive of Formal Proofs. Second, there is the IOA Toolkit [49], which is composed of a formal specification of the IOA language, a simulator [103], a verifier based on the LP theorem prover [34], and a tool for generating Java programs from IOA specifications [36]. 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.

Because many of the existing tools are about a decade old and have not been maintained,

we chose to implement our own theory of I/O automata in Isabelle/HOL. The advantage is that we formalized only what we need, leading to a very simple theory, and we do not depend on unmaintained infrastructure. Our formalization in Isabelle/HOL is presented in section 2.3.5.

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 [67] 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, noted $Acts\left(sig\right)$, is the union of all three components, whereas the set of external actions, noted $Ext\left(sig\right)$, is the union of the inputs and outputs.

State machines

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

- S is the set of states of Σ;
- C is the set of actions of Σ;
- $S_0 \subseteq S$ is the set of initial states of Σ;
- δ is the transition relation of Σ, 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 Σ 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 transitoin, 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 $Acts(A.sig)$, $Start\left(A\right)$ for $Start\left(\Sigma\right)$, $\delta\left(A\right)$ for $\delta\left(\Sigma\right)$, and $States\left(A\right)$ for $States\left(\Sigma\right)$.

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, $\langle s, a, s' \rangle$, whose first state if s. We say that a state is *reachable* if there exists an execution of Σ 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 of 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, whereas the schedule of e does not.

We write $s \stackrel{t}{\Longrightarrow}_A s'$ when there exists an execution fragment $e = \langle s, ps \rangle$ such that last-state $(e) = s'$ and $Trace\,(e) = t$.

Implementation relation

We say that an I/O automaton B *implements* an I/O automaton A, noted $B \leq A$, 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

An sequence of signatures $Sigs$ is said *compatible* when, for every two different indices i, j, the outputs of $Sigs[i]$ and $Sigs[j]$ are disjoint and the internal actions of $Sigs[i]$ and $Sigs[j]$ are disjoint. Note that, in consequence, one cannot compose two identical signatures whose outputs are nonempty.

The composition of a sequence of signatures $\langle Sig_1, \ldots, Sig_n \rangle$, $\prod_{i \in 1..n} Sig_i$, 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 $\langle Sig(A_1), \ldots, Sig(A_n) \rangle$.
- The states of the composition are the sequences $\langle s_1, \ldots, s_n \rangle$ where $s_i \in States(A_i)$ 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.

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 content of the sequence and not on the ordering. 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 2.3.1. *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 2.3.1 (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

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

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

Theorem 2.3.2. *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 2.3.3. *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 and I/O automaton B by adding *history variables* to A, obtaining A_H, and exhibiting a *refinement mapping* from A to B. The technique is an instance of a simulation proof, which include forward simulation, backward simulations, and the use of prophecy variables. In the rest of the thesis we only need history variables and refinement mappings.

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 exists two 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 δ_H is such that

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

Theorem 2.3.4. *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* 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 $f[s] \stackrel{\langle\rangle}{\Longrightarrow}_B f[s']$.

Theorem 2.3.5. *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 2.3.1. *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.*

We will use corollary 2.3.1 throughout the thesis to prove implementation relations between I/O automata.

Forward simulations and backward simulations are other types of simulations that do not require the use of history variables. They are formalized in the Isabelle/HOL theory called "Simulations" which can be found in appendix B.

2.3.5 Isabelle/HOL Formalization

A formalization in Isabelle/HOL of the basics of the theory of I/O automaton can be found in appendix B. The formalization contains the proofs of the three theorems asserting the soundness of refinement mappings, forward simulations, and backward simulations.

2.4 TLA+

In this section we introduce TLA+ informally and we show how to translate I/O automata specification 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 [62] or the article of Merz [77], 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.

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:

$Next1 \triangleq x' = x + 1$

$Init1 \triangleq x = 0$

$Spec1 \triangleq Init1 \wedge \Box Next1$

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 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 $Spec2$.

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

$Next2 \triangleq \wedge y' = \neg y$

$\quad \wedge \text{IF } y \text{ THEN } x' = x + 1 \text{ ELSE } x' = x$

$Spec2 \triangleq Init2 \wedge \Box Next2$

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. Therefore, in TLA+, specifications must be of the form $Init \wedge \Box [Next]_{vars}$, where $Init$ constrains the initial state, $vars = \langle v_1, \ldots, v_n \rangle$ is the list of all the variables appearing in the $Init$ or $Next$ formulas, and $[Next]_{vars}$ is defined as $Next \vee \left(v_1' = v_1 \wedge \cdots \wedge v_n' = v_n\right)$.

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

$Init1 \triangleq x = 0$

$Next1 \triangleq x' = x + 1$

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

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

$Next2 \triangleq \wedge y' = \neg y$

$\quad \wedge \text{IF } y \text{ THEN } x' = x + 1 \text{ ELSE } x' = x$

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

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:

$Inv2 \triangleq x \in Nat \wedge y \in Bool$

THEOREM $Spec2 \Rightarrow \Box Inv2$

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 $Init2 \Rightarrow Inv2$

LEMMA $Inv2 \wedge Next2 \Rightarrow Inv2'$

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 an *refinement proof*. In TLA+, it is formalized as follows.

THEOREM $Init2 \Rightarrow Init1$

THEOREM $Inv2 \wedge Next2 \Rightarrow Next1$

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 $Spec1$ behave the same. To make the observation formal we can hide the y variable of $Spec2$, which we consider internal, using *temporal quantification*.

The specification $Spec2$ becomes

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

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 $Init2 \wedge \Box[Next2]_{\langle x, y\rangle}$.

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 $F1 = Init1 \wedge \Box[Next1]_{vars1}$ and $F2 = Init2 \wedge \Box[Next2]_{vars2}$, 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 $F1 \wedge F2$ 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 $vars2$ are not disjoint, 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. 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 the works of Lamport [62] and Merz [77].

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.

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 noted $[\![A]\!]$ and $[\![B]\!]$, making sure that $[\![A]\!] \Rightarrow [\![B]\!]$ implies $A \leq B$. We assume that A and B have the same external signature. Otherwise we already know that $A \leq B$ does not hold.

We assume that the component 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 define $[\![Sig(A)]\!] = Sig(A)$, $[\![States(A)]\!] = States(A)$, $[\![Start(A)]\!] = Start(A)$, and $[\![\delta(A)]\!] = \delta(A)$.

The TLA+ specification $[\![A]\!]$ uses one variable s_A representing the state of A, one variable

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

and one variable

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

We use the operator

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

to update the variables ext and int, representing the I/O automaton A emitting the action a. We use the flag to distinguish emitting the same action twice from stuttering. The formula $[\![B]\!]$ is defined similarly, uses the same variable ext, but different variable s_B and int_B for the state an internal actions.

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

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

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

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

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 $[\![A]\!] \wedge [\![B]\!]$. This does not work because I/O automata take joint steps only when emitting an action that is common to both I/O automaton. Otherwise they evolve independently. Therefore, when translating A, we separate the ext variable in two variables $common_{AB}$ and ext_A, and when translating B, we separate the ext variable in two variables $common_{AB}$ and ext_B. The three new variables allow A or B to take an independent step, which represents emitting 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. However, 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 a the translation of the transition relation of A in the context B, noted $Next(A)_B$, as follows.

The formula $Next(A)_B$ uses the variables ext_A, $common_{AB}$, int_A, and s_A. Define

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

to update the variables ext_A, whose updates represent emitting an external action that is not

common to A and B, $common$, whose updates represent emitting an external action common to A and B, and int, whose updates represent emitting internal actions of A.

Finally, define

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

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

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

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

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 discussed the choice of a framework for specifying and reasoning about distributed systems and we have presented 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.

Then we have seen that fast prototyping and debugging tools are crucial. Fast prototyping and debugging saves time because it is much faster to discover mistakes in a specification using lightweight debugging tools rather than by trying to attempt a proof. Proof automation obviously speeds up the proof process by allowing bigger gaps to be filled-in automatically in an interactive proof.

To our knowledge, the best tool available for fast prototyping and debugging is the TLA+ Toolbox. However we found the different tools available to mechanically prove properties of TLA+ specification difficult to use.

However, the theory of I/O automata is easily formalized in Isabelle. Therefore, we chose to use both I/O automata and TLA+. We use I/O automaton in the thesis, because their succinct representation of distributed systems, and for mechanically-checked proofs in Isabelle/HOL, because the theory of I/O automata is easily formalized. For prototyping and debugging, we use TLA+.

This choice implied to go back and forth between I/O automata and TLA+ specifications. We have seen how to translate from one to the other, however, with hindsight, we concluded that the overhead incurred by maintaining two different formalizations was not worth it and that we should have invested more time upfront to master the formalization of TLA+ in Isabelle/HOL of Merz [37].

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 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 [43].

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 3.5.1). 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. [32], 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 3.6.1). 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 [61], Herlihy and Wing [43], and Filipovic et al. [32] for more detailed discussions about linearizability and its properties. However, note that these works all rely on the traditional, trace-based, definition of linearizability.

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 β 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* γ, which maps a state and a request to an output. The members of S are called Δ-states.

We say that a data-type representation is *deterministic* when the state machine Σ 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 Δ is *idempotent* when the two following properties hold.

Property 3.2.1. *A duplicate request leaves a Δ-state unchanged: if $Contains\,(r, s)$ holds then $s \bullet r$ equals s.*

Property 3.2.2. *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 3.2.2 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 3.2.1 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 3.2.1 and 3.2.2 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.

In other words, property 3.2.2 states that 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, for every request $\langle q, c' \rangle \in rs$, $q \neq p$, then for every state s,

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

Property 3.2.2 can be also be restated 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 $s \bullet \langle p, c\rangle \bullet \langle q, c'\rangle$,

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

With the fact that duplicate request do not change the state (property 3.2.1), eq. (3.3) implies property 3.2.2.

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 $\gamma\left(1, r_p\right) \neq \gamma\left(3, r_p\right)$. If the transition relation is as represented in fig. 3.2, then Δ violates the second idempotence property because once in state 4, there is no way to 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.

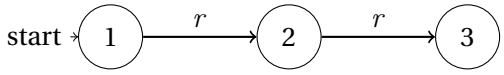

Figure 3.1: A transition relation that violates the first idempotence property (property 3.2.1)

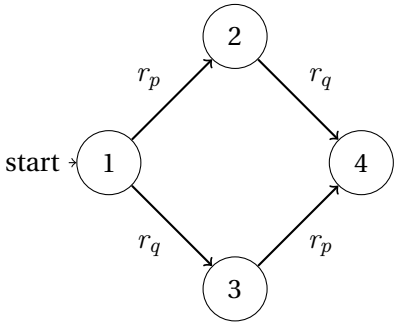

Figure 3.2: A transition relation that violates the second idempotence property (property 3.2.2)

Behaviors

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

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

The data-type representation $\Delta = \langle S, \langle\perp\rangle, C, \Sigma\rangle$ is a data-type representation *of* $D = \left\langle C, O, \beta\right\rangle$ when $Beh(\Delta) = \beta$. Note that a data-type representation uniquely determines a data type but

that a data type may have multiple different representation.

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

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 an 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 the content of the set data structure and, for every client p, of the highest time stamp seen, $ts[p]$, and of the output of 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 δ 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 $ts \leq ts[p]$, the output function γ always returns the value of $last[p]$. Otherwise, if the addition or removal of an element is requested, then true is return, and if the request is of the form $\langle$"contains", $v, ts\rangle$, γ 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 $Cons(V)$ are of the form $\langle$"propose", $v\rangle$ 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 $Cons(V)$ 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 $\{V\} \cup V$, where V means that no value has been chosen yet and $\{v\}$ means that the value v has been chosen. In the initial state, no value has been chosen ($\bot = V$).

The transition relation δ is such that if no value was chosen, then the proposed value is chosen, $\delta(V, \langle$"propose", $v\rangle) = V$, and if a value was already chosen, then the same value is still chosen, $\delta(\{v\}, \langle$"propose", $v'\rangle) = \{v\}$. The transition function δ, when $V = \{v_1, v_2\}$, is represented graphically in fig. 3.3.

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

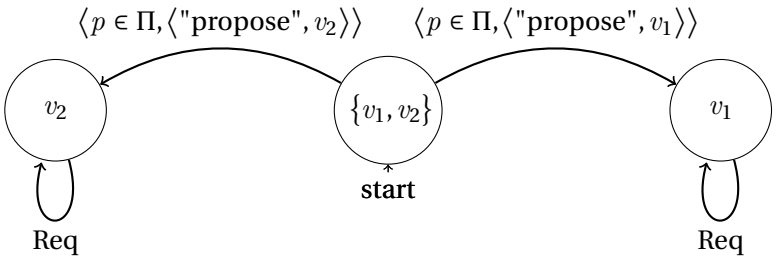

Figure 3.3: The transition relation of a representation of the consensus data type when $V = \{v_1, v_2\}$

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.

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 $\{p\}$ 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 γ returns "Won" in the state Π and "Lost" in all other states.

Note that the $TestAndSet$ data type is idempotent.

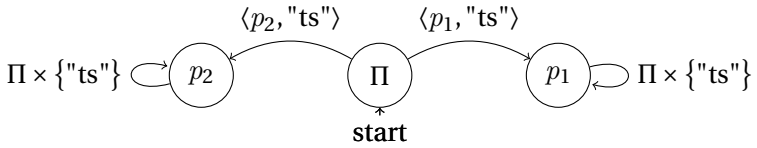

Figure 3.4: The transition relation of a representation of the $TestAndSet$ data type, when $\Pi = \{p_1, p_2\}$

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.

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 $\Delta = \langle\langle S, C, \{\perp\}, \delta\rangle, O, \gamma\rangle$.

To give an idea of the range of possible data-type representations of D we define two representations based on Δ. 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 it has 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 $Unfold(\Delta) = \langle\langle S_1, C, \{\perp_1\}, \delta_1\rangle, O, \gamma_1\rangle$ where S_1 is the set of all histories, Req^*, where the initial state $\perp_1$ 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 Δ, the history s starting from the initial Δ-state, obtaining $\gamma(\perp \star s, r)$.

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 Δ are *output equivalent* if they cannot be distinguished by executing requests and looking at the output produced,

$$s \equiv s' \Leftrightarrow \forall rs \in Req^*, r \in Req : \gamma(s \star rs, r) = \gamma\left(s' \star rs, r\right). \tag{3.5}$$

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 δ' and γ' such that $\delta'\left(Eq(s), r\right) = Eq(\delta(s,r))$ and $\gamma'\left(Eq(s), r\right) = \gamma(s,r)$. The functions δ' and γ' 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 automata 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.

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 Π 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, noted $Proc(a)$, and a command, noted $Cmd(a)$. The invocation of command c by client p is noted $Inv_p(c)$. The set of all invocation actions is noted $Invs$ and the set of all invocation actions of a client p is noted $Invs_p$.

The *response actions* consist of the client which receives the response, noted $Proc(a)$, and of an output, noted $Output(a)$. The response to client p with output o is noted $Resp_p(o)$. The set of all response actions is noted $Resps$ and the set of all response actions of a client p is noted $Resps_p$. 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, noted $t|p$.

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 $p \in \Pi$,

$$Seq = \prod_{p \in \Pi} Seq(p). \tag{3.6}$$

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.

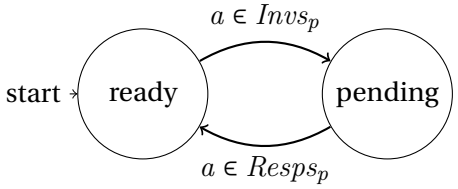

Figure 3.5: The transition relation of the I/O automaton $Seq(p)$.

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 Δ of D such that the projection of A onto the invocation and response actions, noted $\pi_{i/r}(A)$, implements $Lin(\Delta)$.

In fact, as stated in theorem 3.3.1, the set of traces of the I/O automaton $Lin(\Delta)$ is the same for every representation Δ of D. 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 Δ, internally, to determine the output to the requests that it receives. The states of the Lin I/O automaton consist of four components: $dState$, tracking the current Δ-state, and, for every client p, $status[p]$, tracking the control flow location of p, $pending[p]$, containing the pending request of p, and $nxtOut[p]$, 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 updates $status[p]$ to "pending" and additionally updates $pending[p]$ 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 the current Δ-state 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 the current Δ-state, $\gamma(dState, pending[p])$. 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 $nxtOut[p]$.

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 Δ at some point in between the invocation and the response. Therefore, if the operations of two clients p_1 and p_2 overlap, their requests, noted 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.

Theorem 3.3.1. *If Δ_1 and Δ_2 are two representations of D, then $Lin(\Delta_1)$ and $Lin(\Delta_2)$ have exactly the same set of traces.*

Proof sketch. Because any representation of D has the same set of behaviors. □

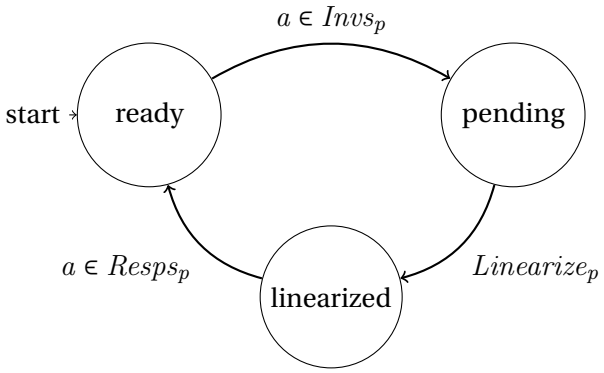

Figure 3.6: Control flow of a client p in the Lin I/O automaton.

3.3.3 Examples: consensus and test-and-set

Consider the Test-and-Set and the Consensus data types that we defined in section 3.2.3. Implementing the I/O automaton Lin(T&S) is equivalent to solving the test-and-set problem, and implementing the $Lin(Cons)$ is equivalent to solving consensus.

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, 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 $Lin\,(Cons)$ 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.

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 Δ-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 $\gamma\big(dState, pending\,[p]\big)$.

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

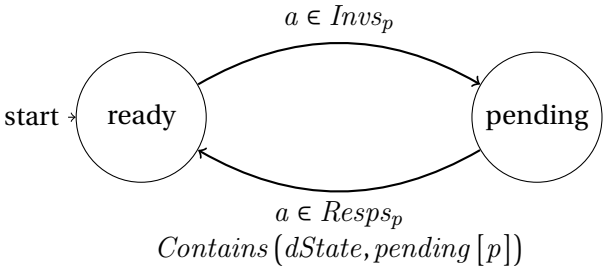

Figure 3.7: Control flow of a client p in the Lin' I/O automaton.

We see that in order to produce a response to the pending request of a client p it is sufficient that the current Δ-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 $nxtOut\,[p]$ component to remember the output that must be returned to the client p. Instead, the Lin' I/O automaton returns $\gamma\big(dState, pending\,[p]\big)$, 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 3.4.1. *The Lin' I/O automaton implements the Lin I/O automaton.*

Proof sketch. 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 it is also ready in t;

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

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

3. if $s.dState$ contains $pending[p]$, then $t.nxtOut[p]$ is the output obtained by executing the pending request of p on $s.dState$.

Note that, for every client p, unless p is in status "pending" or "aborted" and $s.dState$ contains the pending request of p, then $nxtOut[p]$ is unconstrained.

Let us show that f is forward simulation from Lin' to Lin. Assume that s is a reachable state of Lin', that $\langle s, a, s' \rangle$ is a transition of Lin', and that t is a state of Lin such that s, 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 we discuss to states 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 $Invs_p(c)$, we have two sub-cases:

   (a) Assume that $s.dState$ does not contain p's request, $\langle p, c \rangle$. Let $e = \langle t, a, t' \rangle$ where t' is equal to t except that $pending[p]$ 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 $s.dState$ 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 $t'.dState$. However, because $t'.dState$ contains the request of p, executing the request $\langle p, c \rangle$ on $t'.dState$ will leave $t'.dState$ unchanged, by the idempotence property of data-type representations (property 3.2.1). 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 $s.dState$ contains $s.pending[p]$ and that p is in status "pending". Therefore, by definition of f, we have that $t.nxtOut[p] = \gamma(s.dState, \langle p, c \rangle)$ and the status of p in t is "linearized". Thus from t to t' the state is updated as in the $Resp_p(\gamma(t.dState, t.pending[p]))$ 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', the $dState$ is updated to $s.dState \bullet pending[p]$, resulting in $s'.dState$ containing $pending[p]$.

   Suppose that $s.dState$ 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 $s.dState$ does not contain $pending[p]$.

   Therefore any state t' which is related to s' must have $status[p]$ = "commited" and $nxtOut[p] = \gamma(s.dState, s.pending[p])$. 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 $s.dState$ but which is contained in $s'.dState$. Therefore, for every client $q \in Q$, any state t' which is related to s' must have $status[q]$ = "commited" and $nxtOut[q] = \gamma(s.dState, s.pending[q])$. 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

   $$e = \langle t, Linearize_p, t'_0, Linearize_{q_1}, t'_1, \ldots, Linearize_{q_n}, t'_n \rangle \tag{3.7}$$

   where

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

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

   We see that, for every client $q \in Q \cup \{p\}$, q is in status "linearized" in t'_n and $t'_n.nxtOut[q] = \gamma(s.dState, s.pending[p])$. Moreover $t'_n.dState = s.dState \bullet s.pending[p]$. 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 $Linearize_p$ transition of Lin, even though we did not update $dState$: by definition of Q, we know that $t'_0.dState$ contains $pending[q_i]$; therefore, by the idempotence property of data-type representations, executing $pending[q_i]$ on $t'_0.dState$ 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 theorem 3.4.1

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 trivially refines the Lin' I/O automaton, using the identity relation as refinement mapping.

Theorem 3.4.2. *The I/O automaton $NDLin$ implements the I/O automaton Lin'.*

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

3.5 The Abstraction Theorem

The I/O automaton $SeqImp$ is a linearizable implementation of D in which the clients take turns for performing their operations: no two operations overlap. The abstraction theorem (theorem 3.5.1) 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 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 Δ. 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 signature, the same set of states, and the same initial state as the Lin I/O automaton. The $Inv_p(c)$ and $Resp_p(o)$ transitions of $SeqImp$ are also the same as the ones of Lin. The only difference between Lin and $SeqImp$ lies in the $Linearize_p$ transition, which has the same effect as in Lin but is enabled only of 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 3.5.1 (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,*

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

Theorem 3.5.1 casts the result of Filipovic et al. [32] in our framework.

3.6 The Inter-Object Composition Theorem

Consider two data-type representations Δ_1 and Δ_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

$$\Delta = \langle\langle S_1 \times S_2, C_1 \cup C_2, \{\langle\bot_1, \bot_2\rangle\}, \delta\rangle, O_1 \cup O_2, \gamma\rangle \tag{3.8}$$

where, if $c \in C_1$, then $\langle s_1, s_2\rangle \bullet \langle p, c\rangle = s_1 \bullet \langle p, c\rangle$ and $\gamma\big(\langle s_1, s_2\rangle, \langle p, c\rangle\big) = \gamma\big(s_1, \langle p, c\rangle\big)$, and, if $c \in C_2$, then $\langle s_1, s_2\rangle \bullet \langle p, c\rangle = s_1 \bullet \langle p, c\rangle$ and $\gamma\big(\langle s_1, s_2\rangle, \langle p, c\rangle\big) = \gamma\big(s_1, \langle p, c\rangle\big)$.

Theorem 3.6.1 (Inter-Object Composition)**.** *Consider two I/O automata A_1 and A_2. If A_1 implements $Lin(D_1)$ and A_2 implements $Lin(D_2)$, then the composition of A_1 and A_2, $A_1 \times A_2$, implements $Lin(D_1 \times D_2)$.*

Theorem 3.6.1 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 3.6.1 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 $\prec_t$ on the positions of t such that, for all positions i, j, $i \prec_t j$ holds when the operation to which $t[i]$ belongs ends before the operation to which $t[j]$ belongs starts.

For example, if

$$t = \big\langle Inv_p(k_1), Res_p(o_1), Inv_q(k_2), Res_q(o_2)\big\rangle, \tag{3.9}$$

then $1 \prec_t 3$ because the operation to which $Inv_p(k_1)$ belongs ends with $Res_p(o_1)$ at position 2 and the operation to which $Inv_q(k_2)$ belongs starts with $Inv_q(k_2)$ at position 3 (so we take $i' = 2$ and $j' = 3$). Similarly, we also have $2 \prec_t 3$, $1 \prec_t 4$, and $2 \prec_t 4$:

$$\prec_t = \big\{\langle 1,3\rangle, \langle 2,3\rangle, \langle 1,4\rangle, \langle 2,4\rangle\big\} \tag{3.10}$$

However, if

$$t = \big\langle Inv_p(k_1), Inv_q(k_2), Res_p(o_1), Res_q(o_2)\big\rangle, \tag{3.11}$$

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 $i \leq i' < j' \leq j$, $t[i']$ is a response, $t[j']$ is an invocation, $Proc(t[i]) = Proc\big(t[i']\big)$, and $Proc\big(t[j']\big) = Proc\big(t[j]\big)$.

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.

$$t_1 = \langle Inv_p(k_1), Res_p(o_1), Inv_q(k_2), Res_q(o_2) \rangle \tag{3.12}$$

$$t_2 = \langle Inv_q(k_2), Res_q(o_2), Inv_p(k_1), Res_p(o_1) \rangle \tag{3.13}$$

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 σ from the positions of t to the positions of t' such that $t[\sigma[i]] = t[i]$ and σ does not change 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

$$t_3 = \langle Inv_p(k_1), Inv_q(k_2), Res_p(o_1), Res_q(o_2) \rangle. \tag{3.14}$$

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. (3.15)), or $t'|p$ was obtained by appending a response action to $t|p$ (eq. (3.16)).

$$\exists a \in Invs : Append(t'|p, a) = t|p \tag{3.15}$$

$$\exists a \in Resps : t'|p = Append(t|p, a) \tag{3.16}$$

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

Theorem 3.7.1. *For every data-type representation* Δ *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* $Lin(\Delta)$.

Theorem 3.7.1 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".

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 $NDLin$. We have seen that the idempotence property of data-type representations play a crucial role in the correctness of $NDLin$.

We have presented two well-known reduction theorems that simplify linearizability proofs: the inter-object composition theorem and the observational equivalence to a sequential specification. 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 properties 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 only 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.

Building adaptive algorithms ad-hoc is a challenge: it is expensive, not scalable, and forbids incremental design. First, it is expensive and not scalable. Changing mode must 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.

Moreover, instead of synchronizing mode 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 with respect to D.

Therefore, if all the modes of an adaptive algorithm A satisfy the modular property P, then any new mode satisfying P may be added to A without any changes to the existing mode. 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 and the literature contains many different models and experiments.

Pedone [90] shows how optimistic distributed protocols, a notion close to that of speculation, presenting several examples, can boost the performance of distributed systems.

Hiltunen and Schlichting [44] present an informal model for adaptive fault-tolerant system and propose to build adaptive algorithm 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. [16] observe 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 modules that are specialized for particular fault patterns. They apply their ideas to an atomic broadcast protocol, studying in depth the performance of the module 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. [92] and Oreizy et al. [87] study adaptive algorithms that briefly stop servicing requests in order to change mode. Bickford et al. [8] 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 [22, 3] proof assistant.

Chen et al. [19] propose a general model for adaptive systems and an implementation Cactus system. They implement and evaluate an adaptive group communication protocol that continues servicing requests while changing mode. Wojciechowski, Rütti, and Schiper [94, 104, 95] covered extensively 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 and while maintaining the properties of group communication.

McKinley et al. [76] and Oreizy et al. [86] 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 extensively covered by the literature, for example in the works of Rosa et al. [93]

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. [12] propose Dy-BIP, an extension of the BIP framework [7] that supports dynamic addition and removal of components and interactions between components. Attie and Lynch [6] 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 way of abstracting 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

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*, noted $Num(a)$, usually appearing as superscript in action names. For example, an invocation action of instance number i is noted $Inv_p^i(c)$, and $Num\left(Inv_p^i(c)\right) = i$.

For a family 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, $t|p$ 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 the mode 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 current 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*.

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, $t|p$ 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,

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

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

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

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

To compose consecutive mode instances, we will require that, for every $i \in \mathbb{N}$, an 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 defined the I/O automaton $SeqImp$ 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 $ModeInst(i,p)$,

$$ModeInst(i) = \prod_{p \in \Pi} ModeInst(i,p). \tag{4.3}$$

The inputs of $ModeInst(i,p)$ are the init actions of process p, $Switchs_p^i$, and the invocation actions of process p, $Invs_p^i$. The outputs of $ModeInst(i,p)$ 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 $ModeInst(i,p)$ 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, if $i = 1$, then every client is initially ready.

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

1. An init action $Switch_p^i(c)$ is enabled when the client p is idle (possible only if $i = 1$). Its effect is to set the status of the client to "pending".

2. A response action $Resp_p^i(o)$ is enabled when p is in status "pending". Its effect is to set

the status of p to "ready".

3. An invocation action $Inv_p^i(c)$ is enable when p is ready. Its effect is to set the status of p to "pending".

4. An abort action $Switch_p^{i+1}(c, v)$ 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 $ModeInst(i, p)$ stops.

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

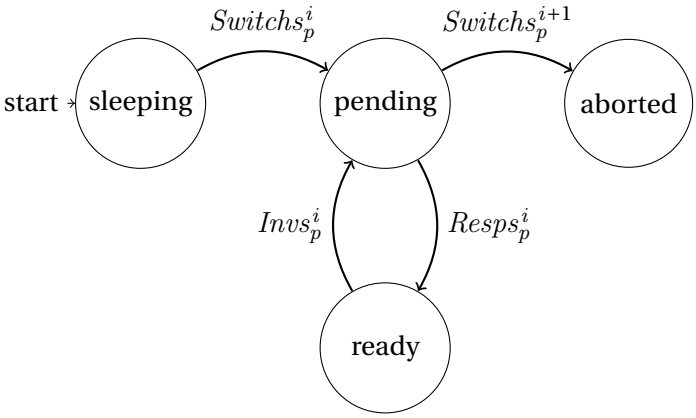

Figure 4.1: The transition relation of $ModeInst(i, p)$, when $i > 1$.

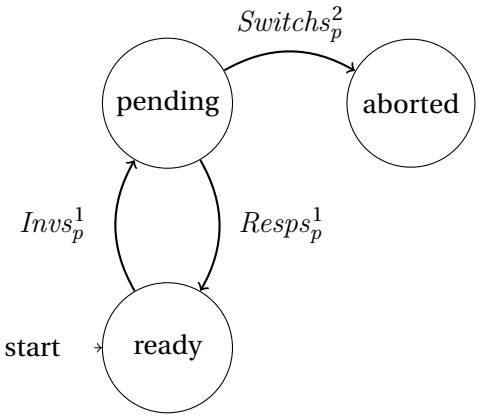

Figure 4.2: The transition relation of $ModeInst(1, p)$.

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

Given a trace t of $ModeInst(i)$, 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 $ModeInst(i)$ for some $i \in \mathbb{N}$ and whose internal actions all have an instance number equal to i. The 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 and the instance numbers, one obtains an I/O automaton whose signature is that of a well-formed data-type implementation.

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

The interface of a 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.

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

$$A = \left(\prod_{p \in \Pi} ModeInst\left(1,p\right) \right) \times \left(\prod_{p \in \Pi} ModeInst\left(2,p\right) \right). \tag{4.4}$$

Applying lemma 2.3.1, we obtain

$$A = \prod_{p \in \Pi} \left(ModeInst\left(1, p\right) \times ModeInst\left(2, p\right) \right). \tag{4.5}$$

For every client p, the states of the I/O automaton $ModeInst\left(1, p\right) \times ModeInst\left(2, p\right)$ are pairs 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 ready in the first mode instance and is 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 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.

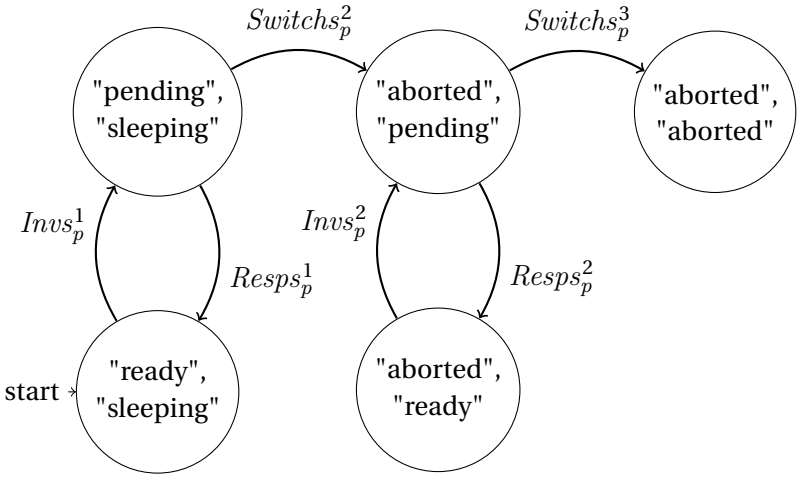

Figure 4.3: The transition relation of $ModeInst\left(1, p\right) \times ModeInst\left(2, p\right)$ where unreachable states have been removed.

Example: Compositing Three Mode Instances

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

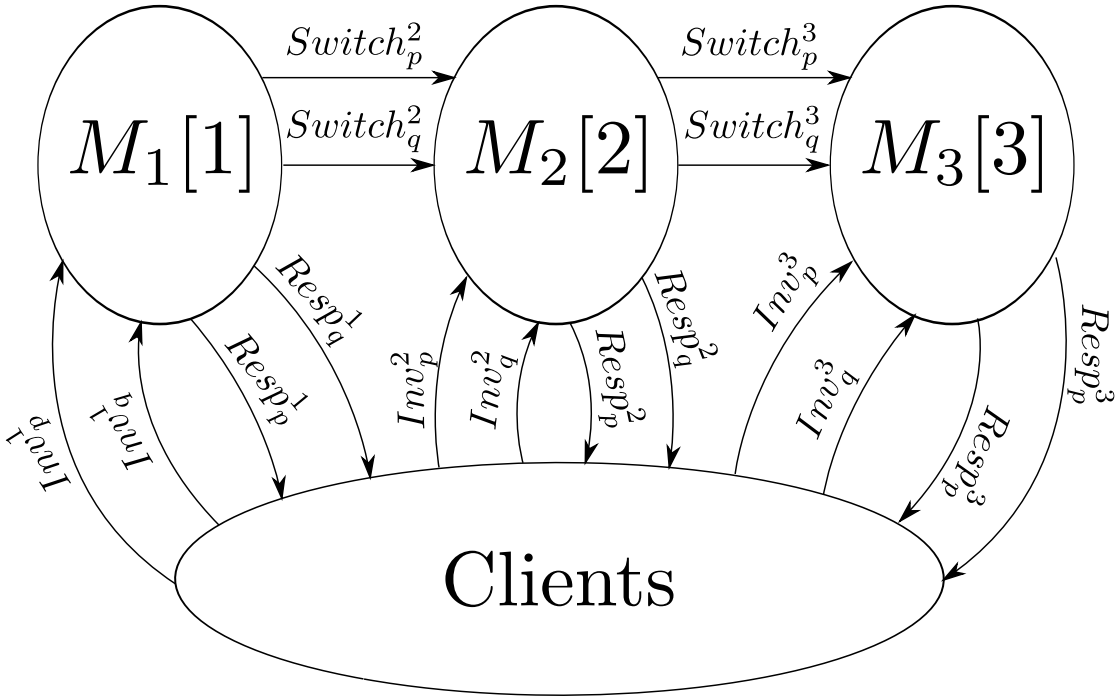

Figure 4.4: Interfaces in a system composed of a three mode instances (that instantiate three different modes M_1, M_2, and M_3), of three processes p, q, and r, and of a client application.

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, modes instance, and we have seen that mode instance can be composed. However, we have not seen exactly how these definition relate to our idea of real adaptive algorithms. 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 a mode 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 processes of a mode instance when to change mode and which mode to change 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 cause problem in her real system. In the algorithm we later present, process can change mode instance at any time, nondeterministically.

We now define a correctness condition for adaptive algorithms and we show that it is sound in the sense that all possible scheduling of modes result in a linearizable execution.

We define the *mode schedules* of an adaptive algorithm A as the I/O automata Sc such that there exists a sequence of modes $\langle M_1, \ldots, M_n \rangle \in A^*$ of modes such that Sc is the product, for

every position i in the sequence, of the i^{th} instance of the mode M_i,

$$Sc = \prod_{i \in Dom(Ms)} M_i[i]. \tag{4.6}$$

We say that the adaptive algorithm A is correct when every mode schedule of A is a linearizable implementation of D, if one ignores the instance numbers of the actions.

Now consider consider a real adaptive algorithm whose modes are modeled by the set of modes A. An execution of a mode schedule corresponds to a run of the adaptive algorithm in which mode instances are scheduled according to their ordered in the sequence. Moreover, for any possible succession of modes observed in a run of a real adaptive algorithm, there is a corresponding mode schedule of A whose traces include the trace of the considered 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 the abstraction is sound.

Note that, by definition of a mode instance, two consecutive mode instances in a mode schedule must synchronize using the init values received (one per process). This is because the init values received is the only information a mode has about the execution of the previous modes. This restriction is in fact an big asset for reasoning modularly about adaptive algorithms, as we will see in the section 4.5.

Finally, the processes may change mode asynchronously. In a correct adaptive algorithm, those asynchronous changes of mode are transparent to the user, which only accesses the system through the invocation and response actions.

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 and 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 $(n+1)^2$ 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 modules, when taken independently of the others. This statement is formalized in the *modularity theorem* below (theorem 4.5.1). With the abstraction theorem (theorem 3.5.1) and the inter-object composition theorem (theorem 3.6.1), the modularity theorem constitutes a third reduction theorem that makes the analysis of adaptive algorithm easier.

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,

$$Invs^{i,j} = \bigcup_{k \in i..j} Invs^k \tag{4.7}$$

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

$$Resps^{i,j} = \bigcup_{k \in i..j} Resps^k; \qquad Switchs^{i,j} = \bigcup_{k \in i..j} Switchs^k. \tag{4.8}$$

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 is between $i+1$ and $j-1$ with bounds included,

$$\pi_{i,j}(A) = hide\left(A, Switchs^{i+1,j-1}\right). \tag{4.9}$$

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

$$\pi_{i/r}(A) = proj\left(A, Invs \cup Resps\right). \tag{4.10}$$

Let P be a family of I/O automata with index set $\mathbb{N} \times \mathbb{N}$,

$$P = \left\{P\left[i,j\right] : i,j \in \mathbb{N}\right\}. \tag{4.11}$$

We say that P is *modular* when

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.

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 $P[1,i+1]$,

$$\pi_{1,i+1}(P[1,i] \times P[i,i+1]) \leq P[1,i+1]. \tag{4.12}$$

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

$$\forall M \in A, i \in \mathbb{N} : M[i] \leq P[i,i+1]. \tag{4.13}$$

4.5.1 The Modularity Theorem

Theorem 4.5.1 (Modularity Theorem)**.** *If P is modular and A satisfies P, then A is correct.*

To prove the modularity theorem, we first need a few lemmas.

Lemma 4.5.1. *If P is modular and $i > 1$, then*

$$Inputs\left(P\left[1,i\right]\right) = Invs^{1,i-1}, \tag{4.14}$$

$$Outputs\left(P\left[1,i\right]\right) = Resps^{1,i-1} \cup Switchs^{2,i}, \tag{4.15}$$

$$Inputs\left(P\left[i,i+1\right]\right) = Switchs^{i} \cup Invs^{i}, \tag{4.16}$$

$$Outputs\left(P\left[i,i+1\right]\right) = Resps^{i} \cup Switchs^{i+1}, \tag{4.17}$$

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

The following corollary of lemma 4.5.1 will be useful in proving theorem 4.5.1:

Corollary 4.5.1.

$$\forall i,j \in \mathbb{N} : \left(\pi_{i/r} \circ \pi_{i,j}\right)\left(P\left[i,j\right]\right) = \pi_{i/r}\left(P\left[i,j\right]\right), \tag{4.18}$$

Proof sketch. By lemma 4.5.1 □

Lemma 4.5.2. *If Ms is a sequence of modes of an adaptive algorithm A and $n = |Ms|$, then*

$$\pi_{1,n+1}\left(\prod_{i \in 1..n} Ms\left[i\right]\left[i\right]\right) = \pi_{1,n+1}\left(\pi_{1,n}\left(\prod_{i \in 1..(n-1)} Ms\left[i\right]\left[i\right]\right) \times Ms\left[n\right]\right) \tag{4.19}$$

Let us now prove the modularity theorem.

Theorem 4.5.1 (Modularity Theorem)**.** *If P is modular and A satisfies P, then A is correct.*

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

$$\forall Ms \in A^{*} : \pi_{i/r}\left(\prod_{i \in Dom(Ms)} Ms[i][i]\right) \leq \text{Lin}(\Delta) \tag{4.20}$$

We proceed by induction on the length of the sequence Ms. Note that we will often implicitly use the monotonicity of composition and projection operators (theorems 2.3.1 and 2.3.3), aons well as lemma 4.5.1.

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

$$IP(Ms) = \pi_{1,n+1}\left(\prod_{i \in 1..n} Ms[i][i]\right) \leq P[1,n+1] \tag{4.21}$$

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

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

Therefore, by corollary 4.5.1,

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

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

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

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

$$\pi_{1,2}\left(\prod_{i \in 1..n} Ms[i][i]\right) = M_1[1]. \tag{4.24}$$

   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 4.5.2,

$$\pi_{1,n+2}\left(\prod_{i \in 1..(n+1)} Ms[i][i]\right) \leq \pi_{1,n+2}\left(\pi_{1,n+1}\left(\prod_{i \in 1..n} Ms'[i][i]\right) \times M[n+1]\right). \tag{4.25}$$

Moreover, by the induction hypothesis,

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

Therefore,

$$\pi_{1,n+2}\left(\prod_{i \in 1..(n+1)} Ms[i][i]\right) \leq \pi_{1,n+2}\left(P\,[1, n+1] \times M\,[n+1]\right). \tag{4.27}$$

Since $M \in A$ and A satisfies P (eq. (4.13)), we get

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

Finally, with the idempotence property of P (eq. (4.12)), we conclude that

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

□

4.6 Conclusion

In this chapter we have motivated the need for adaptive algorithm, which allow building efficient and robust distributed systems. However, we have seen that it is not practical to devise adaptive algorithms in an ad-hoc manner: as the number of possible adaptations grow, the complexity of designing an adaptive algorithm grows quadratically. Morever, incremental design is even more complicated. Therefore, a more principled, modular approach is therefore needed.

We have formalized adaptive algorithms and modular properties, which enable scalable, incremental development of adaptive algorithms.

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 two important questions remain: do modular properties exist and, if yes, are there some which are efficiently implementable in the shared-memory or message-passing models of computation?

In this chapter we answer the first question by proposing a modular property called *speculative linearizability*. Speculative linearizability takes a parameter that allows one to instantiate it for any given data type. We will answer the second question in the next chapter, in which we show that speculative linearizability can be efficiently implemented in the message-passing model.

Given an i^{th} instance, the $SLin(\Delta)[i, i+1]$ automaton models a mode instance, numbered i, in which the processes behave *speculatively*, i.e., they update the state of the system in a way that would work only under optimistic assumptions. If the optimistic assumptions hold, this allows the system to perform efficiently. However, if the optimistic assumptions do not hold, the state of the system can become inconsistent. In this case, the processes must detect the inconsistency and *abort* their execution of the current mode instance and *switch* to the next mode instance, passing it a *switch* Δ*-state*. When the processes 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 instances uses the switch Δ-states received from the previous instance. The family of I/O automata $SLin(A)$ formally specifies this process and, notably, defines how the execution of a mode should be encoded in the switch Δ-states it passes to the next mode, 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 by Lamport [56] to generalize the Paxos algorithm.

5.2 Related Work

Several reduction theorems can simplify the analysis of adaptive distributed algorithms. In the next three paragraph 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 [43, 60, 61, 32] 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; 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.

Elrad and Francez [28] define communication-closed layers and show that to reason about the safety of algorithms composed of communication-closed layers, one does not need to consider the interaction between layers. Charron-Bost and Schiper [18] 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 [31], meaning that verifying them one a system containing 5 processes is sufficient to conclude that the system is correct for any number of processes. Emerson and Kahlon derive cutoff bounds [30] for systems whose process are instances of a generic process template. Examples include a cache coherence protocol. In a later paper [29] they address networks of heterogeneous processes.

The Abstract framework [41] proposes a reduction theorem that is the main inspiration behind the Speculative Linearizability framework. The Abstract Composition Theorem allows to reduce the correctness of an adaptive algorithms to the correctness of its modes taken independently of each other.

5.3 Recoverable Data-Type Representations (RDRs)

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

We say that a state d is smaller than a state d', noted $d \preceq d'$, 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 states is not necessarily a partial order, for example when the transition relation δ has cycles.

A state d is a *lower bound* of a set of 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 states ds, when it exists. Also note that the glb of a set of states does not necessarily exist.

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

Property 5.3.1 (Antisymmetry)**.** *The "smaller than" relation on states, $\preceq$, is antisymmetric.*

Property 5.3.2 (Existence of GLB)**.** *Every two states have a unique greatest lower bound.*

Property 5.3.3 (Consistency)**.** *If the two states both contain a request r, then their glb also contains r.*

Corollary 5.3.1. *Consider three 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 $GLB(d_1, d_2) = d_0 \star rs$.*

Properties 5.3.1 and 5.3.2 imply that that $\langle S, \preceq \rangle$ is a *join semi lattice* with $\perp$ as least element: by definition, $\preceq$ is reflexive and transitive; with property 5.3.1, we get that $\preceq$ is a partial order; with property 5.3.2 we have that $\langle S, \preceq \rangle$ is a join semi-lattice.

We will see that properties 5.3.1 to 5.3.3 are crucial to allow the $SLin$ I/O automaton to recover a consistent state of an RDR given as input a set of different states that were obtained through different executions.

The reader who is familiar with the work of Lamport on Generalized Consensus will recognize the similarity between RDRs and C-Struct Sets. Although similar, RDRs have a notion of behavior that includes the outputs that processes receive, whereas C-Struct Sets do not.

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

Lemma 5.3.1. *Every data type has a recoverable data-type representation.*

Proof sketch. $Unfold\,(\Delta)$ is a recoverable data-type representation of D. □

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

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

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

If b is a behavior of D and i,j are two positions in b, then $Swap\big(b,i,j\big)$ is defined as the behavior b except that the operation at the position i is swapped with the operation at the position j.

We say that two requests r and r' *commute* when, for every behavior b, if r appears at position i immediately followed by r' or if r' appears at position i immediately followed by r, then b is a behavior of D if and only if $Swap(b,i,i+1)$ is a behavior of D.

However, it is often difficult to determine whether two requests commute. Instead, it is easier to use a *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 $\langle r,r'\rangle \in \#$. Thus, the relation # can be seen as an over-approximation of the set of pairs of requests that do not commute.

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 the requests related by the dependency relation #. More precisely, the sequences of requests rs and rs' are equivalent when there exists a permutation σ such that, for every position i, $rs\,[i] = rs'\,[\sigma\,[i]]$ and, for every position j, if $i < j$ and there is a dependency between the request $rs\,[i]$ and the request $rs\,\big[j\big]$, then the permutation σ preserves the order of i and j ($\sigma\,[i] < \sigma\,\big[j\big]$).

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

We now define the transition function $\delta^{\#}$ as mapping 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,

$$\delta_{\#}\left(Eq(rs), r\right) = Eq\left(Append(rs, r)\right). \tag{5.2}$$

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 Δ the request r on the state $\perp \star rs$,

$$\gamma_{\#}\left(Eq(rs), r\right) = \gamma(\perp \star rs, r). \tag{5.3}$$

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 $\gamma_{\#}$,

$$H^{\#}(D) = \left\langle\left\langle H, \{Eq(\langle\rangle)\}, C, \delta_{\#}\right\rangle, O, \gamma_{\#}\right\rangle. \tag{5.4}$$

Note that because Δ is a data-type representation of D, if rs' and rs are equivalent, then, for every request r, $\delta(rs', r)$ and $\delta(rs, r)$ are equivalent and $\gamma(rs, r) = \gamma\left(rs', r\right)$. Therefore γ_H and δ_H are well defined.

We now have the following important property.

Lemma 5.3.2. *If the relation on requests # is a dependency relation of D then the data-type representation $H^{\#}(D)$ is a* recoverable *data-type representation.*

Proof sketch. See section 4.4 of Lamport's paper [56], where the properties of interest are proved in the context of C-Struct Sets. The proof of Lamport is based on the work of Mazurkiewicz [75] on trace theory. □

Lemma 5.3.2 is important because, in contrast to $Unfold(\Delta)$, executing commutative requests in any order always leads to the same state in $H^{\#}(D)$. Therefore, the glb of two states obtained by executing the same set of commuting requests, but in different orders, contains all the requests in the set. With the $unfold(\Delta)$ RDR, the glb would not contain any of the requests. 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

$$SLin = \{SLin[i,j] : i,j \in \mathbb{N}\}. \tag{5.5}$$

Therefore, 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 will first examine the I/O automaton $SLin[1,j]$ where $j > 1$.

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

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

Signature

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

$$Inputs(SLin[1,j]) = Invs^{1,j-1}. \tag{5.6}$$

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

$$Outputs(SLin[1,j]) = Resps^{1,j-1} \cup Switchs^{j}. \tag{5.7}$$

The signature of $SLin[1,j]$ contains all invocation and responses in the instance number range $1..(j-1)$ in order 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, $i,j \in \mathbb{N}$.

The $SLin[1,j]$ 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 $SLin[1,j]$, of the form $Linearize^i$, are actions which linearize a whole sequence of pending requests at once.

State Space and Transition Relation

The state of $SLin\left[1,j\right]$ consists of 4 components, $dState$, tracking the current state of the RDR Δ, $abortVals$, tracking the set of abort Δ-states that have been produced so far, and, for every client p, $status\left[p\right]$, tracking the control flow location of p, and $pending\left[p\right]$, containing the pending request of p.

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

The I/O automaton $SLin\left[1,j\right]$ executes as follows.

1. 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". 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 $Linearize^i$ is similar to the $Linearize$ action of the $NDLin$ I/O automaton, linearizing multiple pending requests at once, but it restricts the possible new Δ-states for $dState$. The action $Linearize^i$ is enabled when at least one client has a pending request. Its effect is to linearize an arbitrary sequence of pending requests rs by updating $dState$ to $dState \star rs$. However, the new Δ-state of $dState$ must be smaller than every abort Δ-state that has been emitted before (i.e. any Δ-state found in $abortVals$).
3. The response action $Resp_p^m\left(o\right)$ where $m \in i..(j-1)$ 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 the current state ofΔ, $o = \gamma\left(dState, pending\left[p\right]\right)$. We say that the Δ-state of $dState$ is a *committed* Δ*-state*, because, in some sense, it has now affected a user of the system. The effect of the response action is to update the status of p to "ready".
4. 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 there exists a sequence of pending requests rs such that the abort Δ-state av is equal to the state obtained by executing rs starting from the current state of Δ ($v = dState \star rs$). The abort action models the client p extracting an "approximate" but safe Δ-state from a implementation that has been corrupted by overly optimistic speculative updates.

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

Important Invariants

Every execution of the $SLin\left[1,j\right]$ I/O automaton satisfies the following important invariants. First, the set $abortVals$ contains all the abort Δ-state produce so far in the execution. Second,

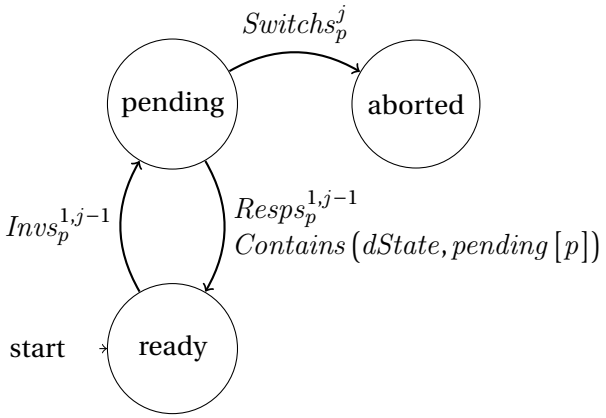

Figure 5.1: The control flow of a process p in the $SLin\,[1,j]$ I/O automaton

abort Δ-states are always larger than committed Δ-states, even if the committed Δ-state appeared only after the abort Δ-state.

Given an execution e of $SLin\,[1,j]$, let the *safe* Δ*-state* at position n be the states of the form $lcv_n \star rs$, where lcv_n is the last committed Δ-state seen in e before position n and rs is a sequence of requests that are pending at position n. Note that the $Linearize^i$ actions update the Δ-state to new Δ-state that is safe at the current position and that is bounded above by every abort Δ-state seen so far.

With the notion of safe Δ-state, we can give a more precise invariant. Every abort Δ-state av is safe Δ-state at the end of the execution e, i.e., there exists a sequence of requests that are pending at the end of e such that $av = lcv \star rs$ where lcv is the last committed Δ-state seen before the end of the execution e.

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

Correctness

We see that, ignoring the abort actions, the actions of the $SLin\,[1,j]$ 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, if one ignores the instance numbers of actions, $SLin\,[1,j]$ implements $NDLin$.

Theorem 5.4.1. *For every $j \in \mathbb{N}$, the projection of $SLin\,[1,j]$ onto the invocation and response actions implements the I/O automaton $NDLin$.*

Proof sketch. Let f be the function mapping a state of s of $SLin\,[1,j]$ to a state t of $NDLin$ such that the $dState$ and $pending$ components of s and t are equal and the status of a client p

in t is the same as the status of p in s except that if $s.status[p]$ = "aborted", then $t.status[p]$ = "pending".

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

Corollary 5.4.1 (Linearizability of $SLin$)**.** *For every $n \in \mathbb{N}$, the project of $SLin[1,j]$ onto the invocation and response actions is linearizable.*

Proof sketch. Using $NDLin \leq Lin$, by transitivity of the implementation relation. □

5.4.2 The I/O Automaton $SLin[i,j]$

According to the definition of a modular property, the composition $SLin[1,j] \times SLin[j,k]$, for $1 < j < k$, must implement $SLin[1,k]$. Therefore, the I/O automaton $SLin[j,k]$ must be able to continue the execution started by $SLin[1,j]$ while preserving linearizability. Moreover, if $k = j+1$, then $SLin[j,k]$ must be a well-formed mode instance. We will now define $SLin[j,k]$ with these constraints in mind.

Signature

We define the signature of $SLin[j,j+1]$ in order for it to be a well-formed mode instance.

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

$$Inputs(SLin[1,j]) = Invs^{j,k-1} \cup Switchs^{j}. \tag{5.8}$$

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

$$Outputs(SLin[j,k]) = Resps^{j,k-1} \cup Switchs^{k}. \tag{5.9}$$

The internal actions of $SLin[j,k]$ are the actions of the form $Linearize_p^j$ and $Recover^j$. The $Recover^j$ action has the task of initializing the $SLin[j,k]$ I/O automaton to a state consistent with the aborted execution of $SLin[1,j]$, using the init Δ-states received. We now discuss how this may be implemented.

State Space

The state of $SLin[1,j]$ consists of 6 components, $dState$, tracking the current state of the RDR Δ, $intVals$, tracking the set of init Δ-states that have been received so far, $abortVals$, tracking the set of abort Δ-states that have been produced so far, $initialized$, a boolean, and, for every

client p, $status[p]$, tracking the control flow location of p, and $pending[p]$, containing the pending request of p.

Initially, $dState$ is $\perp$, the sets $initVals$ and $abortVals$ are empty, $initialized$ is false, and, for every client p, $status[p]$ = "idle" and $pending[p]$ is arbitrary. As in the $ModeInst(i,p)$ I/O automaton, a client p can be either in status "idle", "ready", "pending", or "aborted". Note that, in contrast to $SLin[1,j]$, the initial control flow of a client if not "ready" but "idle".

Recovering $dState$

We have seen that the abort Δ-states of $SLin[1,j]$, which are the init Δ-states of $SLin[j,k]$, are safe Δ-states: every init Δ-state has the form $cv \star rs$, where rs is a sequence of pending requests and cv is the last committed Δ-state of $SLin[1,j]$. Therefore, by corollary 5.3.1, the greatest lower bound of the set of init Δ-states received is also of the form $lcv \star rs$, where rs is a sequence of pending requests and lcv is the last committed Δ-state of $SLin[1,j]$.

Thus, if we initialized the $dState$ component of $SLin[j,k]$ to the glb $lcv \star rs$ of the init Δ-states, we could then run $SLin[j,k]$ as $SLin[1,j]$. The initialization would have the same effect as if $SLin[1,j]$ had linearized the sequence of requests rs.

Moreover, observe that the glb of any nonempty *subset* of the init Δ-states is also of the form $lcv \star rs$, where rs is a sequence of pending requests and lcv is the last committed Δ-state of $SLin[1,j]$. Therefore we will define the $Recover^j$ action of $SLin[j,k]$ as initializing $dState$ to a ***safe init***, defined as the glb of a nonempty subset of the init Δ-states,

$$\begin{aligned} SafeInits = \{ & GLB(is) \star rs : \\ & is \subseteq initVals \wedge rs \in Seq(PendingReqs) \\ & \wedge \forall av \in abortVals : GLB(is) \star rs \preceq av \} \end{aligned} \tag{5.10}$$

where

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

Once the recovery action executed, the I/O automaton $SLin[j,k]$ may proceed exactly as the I/O automaton $SLin[1,j]$, but it needs to wait for the recovery action before it can produce any outputs.

However, we would like our specification of $SLin[j,k]$ to include as many behaviors as possible, in order not to restrict its implementations unduly. Therefore, we will make a few improvements to the $SLin[j,k]$ that we have just described.

Aborting before recovery

Suppose that we allowed the $SLin[j,k]$ I/O automaton to abort before the recovery action. Two problems would arise. First, we need to define what the abort should do when the recovery has not yet taken place: in $SLin[1,j]$, the possible abort Δ-states are defined in terms of $dState$, which is not initialized in $SLin[j,k]$ precisely before the $Recover^j$ action takes place. Second, remember that $SLin[1,j]$ ensures that its abort Δ-states are safe Δ-states, therefore we need to modify the recovery action to make sure that the same invariant holds.

Let us first address the second issue. Since $dState$ can become a committed Δ-state at any point, we modify our definition of *safe inits* by adding the constraint that a safe init be smaller than any abort Δ-state seen so far. Therefore, even if $SLin[j,k]$ aborts before the "recovery" action takes place, we are safe.

To address the first issue, we now define the set of *safe abort* Δ*-states*, $SafeAborts$, ensuring the any safe abort Δ-state can be used as abort Δ-state at any point. Our aim is to ensure that any safe abort Δ-state is a *safe* Δ*-state* at the current point in the execution of the composition $SLin[1,j] \times SLin[j,k]$, i.e., is of the form $lcv \star rs$, where rs is a sequence of pending requests and lcv is the last commit Δ-state of either $SLin[1,j]$ or $SLin[j,k]$. At the same time, we would like the set of safe abort Δ-states to be as big as possible, in order not to restrict speculatively linearizable implementations unduly.

Let G be the set of all the glbs of the nonempty subsets of init Δ-states,

$$G = \{GLB(is) : is \subseteq initVals\}. \tag{5.12}$$

Given a state of $SLin[j,k]$, the set $SafeAborts$ is defined as follows.

1. If the recovery action has not taken place, then the safe abort Δ-states are defined as the glbs of the nonempty subsets of init Δ-states,

$$\neg initialized \Rightarrow SafeAborts = G \tag{5.13}$$

2. If the recovery action has taken place, then there are two cases.

   (a) The Δ-state $dState$ is larger than every member of G. In this case the safe abort Δ-states are the safe Δ-states, i.e., the Δ-states of the form $dState \star rs$ where rs is a sequence of pending requests.

   (b) The Δ-state of $dState$ is smaller than a member of G. In this case we can make $safeAborts$ bigger than in the previous case. The safe abort Δ-states are the Δ-states which are larger than $dState$ and which are of the form $g \star rs$, where $g \in G$ and rs is a sequence of pending requests.

The two cases can be regrouped by the following definition.

$$\begin{aligned} initialized \Rightarrow SafeAborts = \{s : \\ s \preceq dState \wedge \exists rs \in Seq(PendingReqs) : \\ s = dState \star rs \vee \exists g \in G : s = g \star rs\} \end{aligned} \tag{5.14}$$

In every execution of $SLin[1,j] \times SLin[j,k]$, the definition of $SafeAborts$ ensures that every abort Δ-state is of the form $lcv \star rs$, where lcv is the last committed Δ-state in either $SLin[1,j]$ or $SLin[j,k]$ and rs is a sequence of requests that are pending at the current point.

The Complete Transition Relation

The I/O automaton $SLin[j,k]$ executes as follows.

1. The init action $Switch_p^j(c,iv)$ is enabled when p is in status "idle". Its effect is to update $pending[p]$ to $\langle p,c\rangle$, to add iv to the set $initVals$, and to set $status[p]$ to "pending".
2. the $Recover^j$ 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, a member of $SafeInits$, and to set set $initialized$ to true.
3. The invocation action $Inv_p^m(c)$ where $m \in i..(j-1)$ is enabled when p is ready. Its effect is to update $pending[p]$ to $\langle p,c\rangle$ and to set $status[p]$ 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 updating $dState$ to a safe Δ-state that is smaller than every member of $abortVals$.
5. The response action $Resp_p^m(o)$ 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 the current state ofΔ, $o = \gamma(dState, pending[p])$. The effect of the response action is to update the status of p to "ready".
6. The abort action $Switch_p^j(c,av)$ is enabled when p is in status "pending", the pending request of p is $\langle p,c\rangle$, and av is a safe abort, a member of $SafeAborts$.

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

5.4.3 Correctness of $SLin[i,j]$

Theorem 5.4.2 ($SLin$ is Well-Formed)**.** *For every $j \in \mathbb{N}$, $SLin[j,j+1] \leq ModeInst(j)$ and the I/O automata $SLin[1,j]$ and $SLin[j,j+1]$ are compatible.*

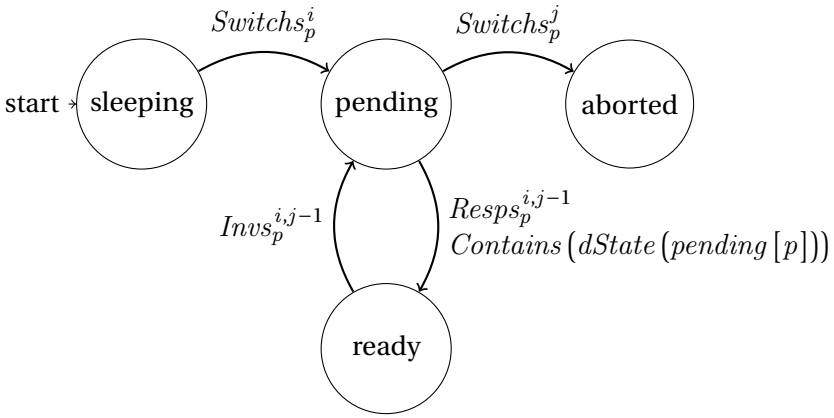

Figure 5.2: The control flow of a process p in the $SLin\left[i,j\right]$ I/O automaton when $i > 1$.

Proof sketch. The function f which maps a state s of $SLin\left[j,j+1\right]$ to the state t of $ModeInst\left(j\right)$ in which the status of every client p is equal to $s.status\left[p\right]$. Then 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, $SLin\left[i,i+1\right] \leq ModeInst\left(i\right)$.

Moreover, 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. □

Lemma 5.4.1. *If* $contains\left(r,s\right)$ *holds, then* $s \bullet r = s$.

Proof sketch. By definition of $contains$ we obtain a sequence $rs \in Req^*$ such that $r \in Image\left(rs\right)$ and $s = \bot \star rs$. With the first idempotence property of the data-type representation Δ (property 3.2.1), we immediately get that $s \bullet r = s$. □

Theorem 5.4.3 (Idempotence of $SLin$)**.** *The family of I/O automata* $\left\{SLin\left[i,j\right] : i,j \in \mathbb{N}\right\}$ *is idempotent.*

To sketch the proof of theorem 5.4.3, we need the following lemmas.

Suppose that $\langle s_1, s_2\rangle$ is a state of $SLin\left[1,j\right] \times SLin\left[j,j+1\right]$.

Lemma 5.4.2 (Invariant 1)**.** *For every member* iv *of the set* $initVals\left(s_2\right)$*, there exists a sequence* rs *of requests that are pending in* s_1*,* $rs \in PendingReqs\left(s_1\right)$*, such that* $iv = dState(s_1) \star rs$.

Corollary 5.4.2 (Invariant 2)**.** *Any safe init* s *of* s_2*,* $s \in safeInits\left(s_2\right)$*, is a safe* Δ*-state of* s_1*, i.e., for every nonempty subset* is *of the init values of* s_2*,* $is \subseteq initVals\left(s_2\right)$*, there exists a sequence* rs *of pending requests of* s_1*,* $rs \in Seq\left(PendingReqs\left(s_1\right)\right)$*, such that* $GLB\left(is\right) = dState\left(s_1\right) \star rs$.

Proof sketch. By lemma 5.4.2, using the consistency property of recoverable data-type representations (corollary 5.3.1). □

Corollary 5.4.2 is crucial to the proof of theorem 5.4.3 because it will allow us to show that a $Recover$ transition of $SLin[1,j] \times SLin[j,j+1]$ is equivalent, under a suitable refinement mapping, to a $Linearize$ transition of $SLin[1,j+1]$.

Theorem 5.4.3 (Idempotence of $SLin$)**.** *The family of I/O automata* $\{SLin[i,j] : i,j \in \mathbb{N}\}$ *is idempotent.*

Proof sketch. We have to show that, for every $j \in \mathbb{N}$ strictly greater than 1,

$$\pi_{1,j+1}(SLin[1,j] \times SLin[j,j+1]) \leq SLin[1,j+1]. \tag{5.15}$$

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

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

The function f is a refinement mapping from $SLin[1,j] \times SLin[j,j+1]$ to $SLin[1,j+1]$. The proof is quite lengthy and technical, therefore we do not include it here and we instead point the reader to our Isabelle/HOL proof. The Isabelle/HOL proof proves the result assuming that the data type is the $Generic$ data type described in section section 3.2.3.

Let us just examine the case of the $Recover^j$ action.

Assume that $\{s_1, s_2\}$ is a reachable state of $SLin[1,j] \times SLin[j,j+1]$ and that $\langle \{s_1, s_2\}, Recover^j, \langle s_1', s_2' \rangle \rangle$ is a transition of $SLin[1,j] \times SLin[j,j+1]$. Let us show that there exists an execution fragment e whose first state is $f[\langle s_1, s_2 \rangle]$, whose last state is $f[\langle s_1', s_2' \rangle]$, and such that the trace of e in $SLin[1,j+1]$ is the empty sequence.

By definition of the $Recover$ action, the boolean $initialized(s_2)$ is false, $initialized(s_2')$ is true, and $dState(s_2')$ is a safe init.

According to the definition of f, from $f[\langle s_1, s_2 \rangle]$ to $f[\langle s_1', s_2' \rangle]$, the $dState$ component changes from $dState(s_1)$ to $dState(s_2')$, while all the other components remain unchanged.

By corollary 5.4.2, we know that $dState(s_2')$ is a safe Δ-state at $f[\langle s_1, s_2\rangle]$. Therefore, $e = \langle f[\langle s_1, s_2\rangle], Linearize, f[\langle s_1', s_2'\rangle]\rangle$ is a $Linearize$ transition of $SLin[1, j+1]$. Moreover, by definition of the signature of $SLin[1, j+1]$, the trace of e in $SLin[1, j+1]$ is empty. □

The refinement mapping used in the proof of theorem 5.4.3 is formalized in TLA+ in appendix A. It is the same as in the proof above.

Theorem 5.4.4. *The family of I/O automaton* $\{SLin[i, j] : i, j \in \mathbb{N}\}$ *is a modular property.*

Proof sketch. Theorem 5.4.2 shows that $SLin[i, i+1]$ is a well-formed i^{th} mode instance, corollary 5.4.1 shows that $SLin[i, i+1]$ is linearizable, and theorem 5.4.3 shows that $SLin[i, i+1]$ is idempotent. Therefore $\{SLin[i, j] : i, j \in \mathbb{N}\}$ is a modular property. □

5.5 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 RDR in a way that leads to increased performance under some optimistic assumptions and to the corruption of the state of the system otherwise. If the state of the system is corrupted by an overly optimistic update, then the processes must detect it, abort their execution, and switch to the next mode, bringing along their estimate of the corrupted 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.

In the next chapter we will see that the speculative linearizability property is efficiently implementable in the message-passing and shared-memory models of computation. To do so, we will present adaptive algorithms that satisfy speculative linearizability and that efficiently implement any data type.

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 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 a representation Δ of D. 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 processes can just use another one.

SMR works but has a drawback: because the requests are ordered by independent consensus instances, SMR 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 processes in Π.

The notion of Generalized Consensus [56], abbreviated GC, allows one to solve this problem. GC formalizes the task of agreeing, modulo the order of commuting requests, on a growing sequence of requests. Therefore GC is a specification of the problem that SMR is trying to solve, except that it relaxes the requirements for commuting requests. In contrast to SMR, GC does

not mandate a specific implementation technique. In fact, SMR can be viewed as a possible implementation of GC, albeit one that does not take advantage of commutativity. In GC, the parties agreeing on the sequence of requests are the processes Π, and not the servers. The servers are now part of the implementation of GC and need not follow any specific protocol. Therefore, in contrast to SMR, there is no artificial separation between consecutive requests. GC is similar to linearizability but abstracts over how the processes should compute outputs, instead focusing on how to learn about the Δ-state.

Generalized Paxos, abbreviated GP, is an adaptive algorithm in the spirit of Fast Paxos [55] which implements Generalized Consensus. The servers of GP, 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". 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 GC allow a fast ballot to process non-conflicting requests with a latency of twice the communication delay without relying on a leader process. In contrast, Paxos has only one type of ballots with a latency of more than three communication delays and relies on a correct leader process. A classic ballot of GP is similar to a ballot of Paxos. The two types of ballots of GP can be seen as two modes of an adaptive algorithm.

However, ballots do not have a clear interface like mode instances, GP has only two types of ballots, and adding new ballot types is not easy. Multicoordinated Paxos [13] is an optimization GP which adds a new ballot type. The specification of Multicoordinated Paxos in TLA+ is more than 10 pages long [13]. Moreover, Multicoordinated Paxos is the only instance of optimization of GP that we know of, perhaps owing to the fact that, although Paxos is already notoriously hard to understand, GP is even more intricate than Paxos. In conlusion, GP 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.

In fact, to prove Quorum and ZLight correct, we propose two more abstract 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 [40]. 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. Moreover, only a strict majority quorum is required for aborting. 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 fast mode in the spirit of Multicoordinated Paxos [13].

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.[38]. Quorum is optimized for the execution of non-conflicting requests and can withstand 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 do not commute and can withstand 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 $H^{\#}(D)$.

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 was. On top of the client processes, we consider a set of N *acceptor* processes.

6.2 Related Work

There are many fault-tolerant algorithms that 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 [73], Multi-Ring Paxos [72], Fast Paxos [55], Disk Paxos [33], Egalitarian Paxos [81], Multi-Coordinated Paxos [13], Vertical Paxos [64], Cheap Paxos [65], Paxos-MIC [48], Mencius [71], and Fast Mencius [101]. In the Byzantine model, examples of algorithms based on Paxos include FaB Paxos [74], Zyzzyva, [53], PBFT [15], Aardvark [21], Q/U [1], and HQ [24].

The Abstract framework [41] 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. The speculative linearizability framework, presented in chapter 5, is inspired from the Abstract framework.

Generalized Paxos [56] 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 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.

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 acceptor 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 automatonand ZLight refines the safe mode I/O automaton.

The two I/O automata $Safe(i)$ and $Fast(i)$ have the external signature of a mode instance i and have the same set of states, namely the states of $SLin[i, i+1]$ except that, for every accepto a, two component $accStatus[a]$ and $dState[a]$ are added.

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 four types of internal actions: $Panic(c)$, where c is a client, and $Exec(a)$, $WakeUp(a)$, and $Stop(a)$, for an acceptor a. Like in the $SLin(i, i+1)$ I/O automaton, we describe the control flow of the processes informally. Clients are either idle, ready, have a pending request, or have aborted, except that they can also have panicked. After a $Panic(c)$ action and before c aborts, we say that c has panicked. As in $SLin(i, i+1)$, if $i = 1$, then every client is initially ready; otherwise, every client is initially sleeping. The acceptors are 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 $\perp$.

6.3.1 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.

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

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 ensured by a leader, as in ZLight, but other implementations are possible.

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

- An init action $Switch_c^i(k, v)$ is enabled when the client c is not initialized, which is possible only if $i > 1$. The action adds v to the set $initVals$ and sets $pending[c]$ to $\langle c, k\rangle$.
- An invocation action $Inv_c^i(k)$ is enabled when the client c is ready. The action sets $pending[c]$ to $\langle c, k\rangle$.
- $WakeUp(a)$, executed by an acceptor a, is enabled if a is idle and if there exists $s \in initVals$ such that substituting s for the value of $dState[a]$ would leave the acceptors in a consistent state. The effect of the action is to perform the substitution.
- $Exec(a)$ is enabled when a is ready, if a client c has a pending request $\langle c, k\rangle$, and if substituting $dState[a] \bullet k$ 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.
- A response action $Resp_c^i(o)$ is enabled when the client c 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 $\sqcup S_Q$ contains r and $o = \gamma(\sqcup S_Q, r)$.
- $Panic(c)$, executed by a client c, is enabled when c has a pending request. The action models the client c detecting that the acceptors have inconsistent Δ-states or the client c not receiving answers from any quorum of acceptors for a too long time.
- $Stop(a)$, executed by an acceptor a, is enabled when there is a client c which has panicked. The action models the acceptor a receiving through the network a notification that the client c has panicked. At this point the acceptor a sends its Δ-state in an acknowledgement to c and stops accepting any new requests (it may already have done so).
- The abort action $Switch_c^{i+1}(k, v)$ is enabled when c has panicked, $pending[c] = \langle c, k\rangle$, and there exists a safe quorum Q of acceptors which have stopped and whose set of Δ-states S_Q is such that $Max(S_Q) = v$. The action models c receiving from every acceptor in $a \in Q$ an acknowledgement that a has stopped along with the Δ-state of a.

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 it. Then $Safe(i)$ refines $SLin(i, i+1)$ under the following refinement mapping. Let every client c have the same status as in $Safe(i)$, except that when c has panicked in $Safe(i)$, then c is considered pending in $SLin(i, i+1)$. Let $initialized$ be true if and only if the is a safe quorum of acceptors which are not idle. Let the Δ-state $dState$ of $SLin(i, i+1)$ be the maximum over all safe quorums Q of the glb of the Δ-states of the members of Q:

$$s = Max\left\{\left\{dState[a] : a \in Q\right\} : Q \in SafeQuorum \wedge \forall a \in Q : status[a] \neq idle\right\}. \tag{6.2}$$

Finally, let $initVals$ and $abortVals$ be unchanged.

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 an extension with pending requests of the global Δ-state $dState$ of $SLin(i, i+1)$. Before the action, we know that the global Δ-state $dState$ is the glb of the Δ-states of a safe quorum Q of acceptors. Therefore, every acceptor of Q has a Δ-state greater than or equal to $dState$. By property of safe quorums, any other safe quorum R has a member $b \in Q$. Moreover, because the acceptors are always consistent, every acceptor $a \in R$ is such that $dState[a]$ is a prefix of $dState[b]$ or vice versa. Therefore the maximum m of the Δ-states of R is an extension of $dState$. Finally, the acceptors only execute pending requests, so the m is an extension of $dState$ 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 The $Fast(i)$ I/O automaton

To compute the output to its request, a client of the $Fast(i)$ I/O automaton communicates with a *fast quorum* of acceptors. In contrast to the safe quorums of the $Safe(i)$ I/O automaton, the Δ-states of a fast quorum of acceptors are allowed to become inconsistent, allowing implementations that respond to a client 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 implementations to eventually abort. 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:

$$|Q \cap R| \geq \left\lfloor \frac{|R|}{2} \right\rfloor + 1. \tag{6.3}$$

Fast quorum and recovery quorum have been described before in the context of Refined Quorum Systems [40] and to derive lower bounds on asynchronous consensus [58]. To satisfy the constraints on the intersection of quorums, we can take the following definition of fast and recovery quorums:

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

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

or

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

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

or

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

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

The actions of the $Fast(i)$ I/O automaton are similar to the ones of the $Safe(i)$ I/O automaton. The $Exec(A)$, $Resp(a)$, and $WakeUp(A)$ actions are identical to the ones of the $Safe(i)$ 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.

In this situation, the abort action has to be changed more dramatically. Taking the maximum of the Δ-states of a fast quorum of stopped acceptors is not safe: because acceptors may be inconsistent, one acceptor may have a very large Δ-state that is completely unrelated to the Δ-states that were used to compute responses. We need to find a Δ-state that is an extension with pending requests of the last commit Δ-state lcv. We know that lcv is the glb of the Δ-states of a fast quorum Q. Therefore, every member of $a \in Q$ has a larger than lcv:

$$\forall a \in Q : dState[a] \geq l \tag{6.10}$$

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

$M = R \cap Q$, then l_M is an extension of l. Therefore, the maximum, over all strict majorities M of R, of the glb l_M, is an extension of l. Since acceptors only execute pending requests, it is also an extension of l with pending requests. To conclude, the abort action of $Fast(i)$ is the same as the abort action of $Safe(i)$ except that there exists a recovery quorum R such that the abort value $av(R)$ is the maximum over every strict majority subset M of R of the glb l_M of the Δ-states of the acceptors in M:

$$av(R) = Max\left\{\sqcup\{dState[a] : a \in M\} : M \subseteq R \wedge |M| \geq \left\lfloor \frac{|R|}{2} \right\rfloor + 1\right\} \tag{6.11}$$

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.

We have proved that both the safe and fast modes are speculatively linearizable. Therefore, any concrete mode refining either the safe or fast modes is also speculatively linearizable and can be combined with any other speculatively linearizable mode. 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 ZQ 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 $QZ = \{Quorum, ZLight\}$. 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 same 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.

6.4.1 Quorum

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 $RcvExecAck(c)$ and $RcvPanicAck(c)$.

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 c, of two arrays $execAcks[c]$ and $panicAcks[c]$ mapping every acceptor a to a Δ-state. Moreover, the state has a network component that we will not explicitly describe. However, the network allows any client or acceptor to send or receive

messages to other clients or acceptor. Remember that the state of $Fast(i)$ has the following components: For every client or acceptor p, $status[p]$ is the control flow location of p; for every client c, $pending[c]$ contains the pending requests of c 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. The initial states are the same as those of $Safe(i)$ except that, for every client c, $execAkcs[c]$ and $panicAcks[c]$ map every acceptor to the special value $none$:

We now describe that actions of the $Quorum(i)$ I/O automaton.

- An init action $Switch_c^i(k,v)$ is enabled when the client c is not initialized, which is possible only if $i > 1$. The action adds v to the set $initVals$, sets $pending[c]$ to $\langle c,k\rangle$, and broadcasts the messages $\langle$"init", $v\rangle$ and $\big\langle$"req", $\langle c,k\rangle\big\rangle$ to all the acceptors.
- An invocation action $Inv_c^i(k)$ is enabled when the client c is ready. The action sets $pending[c]$ to $\langle c,k\rangle$ and broadcasts the message $\big\langle$"req", $\langle c,k\rangle\big\rangle$ to all the acceptors.
- $WakeUp(a)$, executed by an acceptor a, is enabled if a is idle and a can receive an $\langle$"init", $v\rangle$ message from a client. The effect of the action is to receive the message and set $dState[a]$ to v.
- $Exec(a)$ is enabled when a is ready and a can receive a $\big\langle$"req", $\langle c,k\rangle\big\rangle$ message from a client. The effect of the action is to receive the message, set $dState[a]$ to $dState[a]\bullet\langle c,k\rangle$, and send the message $\langle$"execAck", $dState[a]\bullet\langle c,k\rangle\rangle$ to c.
- $RcvExecAck(c)$ is enabled when the client c can receive a message $\langle$"execAck", $v\rangle$ from an acceptor a. Its effect is to receive the message and to set $execAcks[c][a]$ to v.
- A response action $Resp_c^i(o)$ is enabled when there exists a fast quorum Q of acceptors such that, for every $a \in Q$, c received an ack from a, the glb $g = \sqcup\{execAcks[c][a] : a \in Q\}$ of the acks contains the pending request of c, and $o = \gamma(g, pending[c])$.
- $Panic(c)$, executed by a client c, is enabled when c has a pending request. Its effect is to broadcast the message $\big\langle$"panic"$\big\rangle$ to all the acceptors.
- $Stop(a)$, executed by an acceptor a, is enabled when a can receive a $\big\langle$"panic"$\big\rangle$ message from a client c. Its effect is to receive the message, stop a, which will not execute any more requests, and to send the message $\big\langle$"panicAck", $dState[a]\big\rangle$ to c.
- $RcvPanicAck(c)$ is enabled when c has panicked and can receive a $\big\langle$"panicAck", $v\big\rangle$ message from an acceptor a. Its effect is to receive the message and to set $panicAcks[c][a]$ to v.
- The abort action $Switch_c^{i+1}(k,v)$ is enabled when c has panicked, $pending[c] = \langle c,k\rangle$, and there exists a recovery quorum R of acceptors such that

$$v = Max\left\{\sqcup\{panicAcks[c][a] : a \in M\} : M \subseteq R \wedge |M| \geq \left\lfloor \frac{|R|}{2} \right\rfloor + 1\right\} \tag{6.12}$$

Equation (6.12) comes from eq. (6.11), which is 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 for a small system size using the Consensus and Generic data types.

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 will 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 can still process them with a latency of two communication delays. This is because executing two commuting requests always results in the same state, whichever the order of their execution.

6.4.2 ZLight

The signature and the states of the $ZLight\,(i)$ I/O automaton is the same as the one of the $Quorum\,(i)$ I/O automaton. Their actions differ 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 $leader$. The actions of the $ZLight\,(i)$ I/O automaton are obtained by modifying those of the $Quorum\,(i)$ I/O automaton as follows.

- In an init action $Switch_c^i\,(k, v)$, the client c sends its $\langle$"init", $v\rangle$ and $\big\langle$"req", $\langle c, k\rangle\big\rangle$ messages only to the leader, instead of broadcasting to all the acceptors.
- In an invocation action $Inv_c^i\,(k)$ the client c also sends its $\big\langle$"req", $\langle c, k\rangle\big\rangle$ message only to the leader, instead of broadcasting it to all the acceptors.
- $WakeUp\,(leader)$ is as in $Quorum$ (the leader is also an acceptor) except that, on top of sending an acknowledgements to the client, the leader broadcasts the message $\langle$"leader-init", $v\rangle$ to all the other acceptors.
- The $Exec\,(leader)$ action is as in $Quorum$ except that, on top of sending an acknowledgements to the client c, the leader broadcasts the message $\big\langle$"leader-exec", c, $dState'\,[leader]\big\rangle$ to the other acceptors, where $dState'\,[a]$ is the new Δ-state of the leader.
- $WakeUp\,(a)$, where a is not the leader, is enable 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.

- $Exec(a)$, where a is not the leader, is enabled when a is ready and a can receive a $\langle$"leader-exec", $c, v\rangle$ message from the leader. The effect of the action is to receive the message, to set $dState[a]$ to v, and to send the message $\langle$"execAck", $v\rangle$ to c.
- $RcvExecAck(c)$ is exactly as in $Quorum$.
- A response action $Resp_c^i(o)$ is as in $Quorum$ except that a safe quorum is substituted for the fast quorum.
- $Panic(c)$, $RcvPanicAck(c)$, and $Stop(a)$ are exactly the same as in $Quorum$.
- The abort action $Switch_c^{i+1}(k, v)$ is enabled when c has panicked, $pending[c] = \langle c, k\rangle$, and there exists a safe quorum R of acceptors such that v is the maximum Δ-state among the Δ-states of the acceptors in R.

$ZLight(i)$ 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 Δ-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 Δ-states.

The refinement mapping has been checked by TLC for a small system size using the Consensus and Generic data types.

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 the same progress guarantees as QZ.

6.5 Speculatively Linearizable Generalized Paxos

In this section we informally show how to modify Generalized Paxos to make it speculatively linearizable. Therefore one can compose Generalized Paxos with QZ. This could be useful in case of hight leader turnover. In Generalized Paxos, a faulty leader is replaced by changing ballot, which may or may not be faster than switching ZLight instance.

We will use the terminology defined in the paper "Generalized Consensus and Paxos" [56]. To understand this section, the reader must be familiar with the abstract Generalized Paxos algorithm, as described in section 5.3 of Lamport's paper [56]. We assume that each client in Π plays the role of both a proposer and a learner.

Generalized Paxos can be thought of as a linearizable implementation of the data type D, represented by Δ. In GP, processes learn about the evolution of the state of Δ, but the model of Lamport does not specify how to compute outputs. Once a process learns a new state, it may to compute the output to its request by checking whether its request is contained in the new state and, if it does, use the output function γ to determine the output. Otherwise the process waits to learn another state in which its request may be contained. Note that this relies on the idempotence property of data types to work correctly, as is the case in the $SLin$ I/O automaton, in which once a state is recovered from the init values, the processes need to determine whether their request is contained in the state and what is the corresponding output.

However Generalized Paxos cannot abort or be initialized. It is not a *mode*. To make it a mode in the speculative linearizability framework, we first modify it to allow initialization. Let "invalid" be a special value which is not a command nor the value "none". We initialize the ballot array as in Generalized Paxos except that for every acceptor a, $\beta_a[0]$ = invalid instead of $\perp$. Then we modify the acceptors so that they wait for a 0-Quorum of acceptors a to have $\beta_a[0] \neq$ invalid. We modify the proposers such that upon an init action, the proposers broacdast their init value to all the acceptors. We modify the acceptors so that when $\beta_a[0]$ = invalid, the acceptor a sets $\beta_a[0]$ to the first init value that it receives. Then the acceptor queries the other acceptors a' to check the value of $\beta_{a'}[0]$. When a Quorum of acceptors is such that $\beta_a[0] \neq$ invalid, then the querying acceptor can proceed executing GP normally.

For performance, it is also useful to allow Generalized Paxos to abort and switch to a more efficient mode like Quorum or ZLight. To enable Generalized Paxos to abort, we add a special "abort" command to the data type. A proposer who wishes Generalized Paxos to abort simply proposes the "abort" command. We modify the acceptors so that when $\beta_a[n]$ contains the

"abort" command, then the acceptor broadcasts $\beta_a[n]$ to all the learners and stops accepting new commands. We modify the learners (which are also the acceptors in our setting) so that upon receiving $\beta_a[n]$ from an acceptor that stopped, a learner switches to the next mode instance using $\beta_a[n]$ as abort value.

By applying the modifications described above, we conjecture that we obtain a speculatively linearizable version of Generalize Paxos, which can therefore be combined as-is with Quorum, ZLight, or any other speculatively linearizable mode, to build an adaptive algorithm implementing the data type D. This is only a conjecture because the author did not have the time to specify the modifications formally and model-check the resulting algorithm. It is thus certain that the above description is too vague and that the details are wrong, but it conveys an important intuition.

6.6 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 quorum of 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. Moreover, we have 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 is useful 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 register [42]. However the paper of Luchangco et al. [66] presents an adaptive consensus algorithm which uses only 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 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 [25]. 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 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 an response or switch action (lines 8, 10, 17, 19, 23 of fig. 7.1, lines 7, 11, and 13 of fig. 7.2), then it stays 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. Observe that the following property is an inductive invariant: if $winner$ has been set, then for every other client p, if $winner \neq p$ and $X = p$, then p has not reached past line 8. When $winner$ is first set, we have $X = winner$ and $Y = true$. For another client $p \neq winner$ to set X to p, p 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. Therefore, if one 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 $\perp$, 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 correcntess of $RegCons$ using the following refinement mapping.

Theorem 7.0.1. *The mode $RegCons$ is a speculatively linearizable first instance.*

Proof sketch. 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 map 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 the 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 $\{RegCons, CasCons\}$.

```
1: Algorithm RCons_p
2: Shared Δ-state dState, initially ⊥
3: Shared boolean decided, initially false
4: Shared boolean contention, initially false
5: Function Invoke_p^1(val):
6: if decided = true then
7:    if contention = false then
8:       Return_p^1(dState)
9:    else
10:      Switch_p^2(val, dState)
11:   end if
12: end if
13: if Splitter(p) = true then
14:   dState ← val
15:   if contention = false then
16:      decided ← true
17:      Return_p^1(val)
18:   else
19:      Switch_p^2(val, ⊥)
20:   end if
21: else
22:   contention ← true
23:   Switch_p^2(val, dState)
24: end if
```

Figure 7.1: The $RegCons$ Mode

```
1: Algorithm Splitter
2: Shared boolean Y, initially false
3: Shared process id X
4: Function Splitter(p):
5: X ← p
6: if Y = true then
7:    return false
8: end if
9: Y ← true
10: if X = p then
11:   return true
12: else
13:   return false
14: end if
```

Figure 7.2: The Splitter Algorithm

```
1: Algorithm CasCons_p
2: Shared Δ-state dState, initially ⊥
3: Function Switch_p^2(val, sval):
4: CAS(dState, ⊥, sval)
5: Response_p^2(dState)
```

Figure 7.3: The $CasCons$ Mode

8 Mechanically-Checked Proofs

8.1 Related Work

8.1.1 Mechanically-Checked Proofs in TLA+ and Isabelle/HOL

Olaf Müller formalized the theory of I/O automata, including liveness, in Isabelle/HOL [82]. Using their framework, they conduct a case study on an industrial helicopter alarm system.

Mauro Jaskelioff and Stephan Merz proved the correctness of Disk Paxos in Isabelle/HOL [50]. They formalize the algorithm as a relation on a set of states and prove its safety properties. Taken together, the formalization and the proofs sum up to roughly 7000 lines.

Recently, Leslie Lamport proved the correctness of a Byzantine Paxos algorithm [59]. The algorithm is formalized in TLA+. The proof consists of three refinement steps and proves the safety of the algorithm. The two most difficult refinement steps have been checked with TLAPS [23], while the other step was checked with the TLC model-checker [105].

Finally, Debrat and Merz [26, 17] formalized in Isabelle/HOL six different consensus algorithms in the Heard-Of Model and proved them correct, including liveness, in Isabelle/HOL. The Heard-Of Model provides a reduction theorem that helps analyzing round-based distributed algorithms. The authors estimate, using their experience proving Disk Paxos, that using the Heard-Of model reduces the length of the proofs by one order of magnitude.

8.1.2 Proving Linearizability

We have endeavored to prove linearizability by refinement, formalizing our work in Isabelle/HOL. Schellhorn et al. employ a similar approach [96], using the KIV theorem prover [27]. Automated techniques are proposed by Vafeiadis [100] and Amit et al. [5]. Finally, O'Hearn et al. study proving the linearizability of wait-free shared-memory algorithms [85].

8.2 Isabelle/HOL Formalization of Speculative Linearizability

8.3 Personal Experience of the Author with Isabelle/HOL

Writing proofs that can be mechanically checked is a challenging topic subject to ongoing research [54, 17]. Therefore, demonstrating that our work simplifies writing mechanically checked proof would provide strong evidence of its power to simplify the design and analysis of distributed systems.

There are two major constraints that guide our choice of a modeling language.

1. We would like to represent *concisely* our objects of discussion, namely systems built out of several components which may interact by performing a *discrete joint action* and otherwise evolve completely *asynchronously*.
2. We would like *mechanically-checked proofs* to be as easy as possible.

There are a few broad features that we can use to classify specification frameworks.

- Most languages can be loosely classified as based on automata theory or based on a kind of process algebra. We will concentrate on languages based on automata theory, for lack of knowledge about process algebra: we chose to save for other work the effort needed to understand their specific advantages and disadvantages.
- Some languages have corresponding software tools for writing and displaying specifications, and checking and proving their properties. These software tools are akin to programming IDEs. As we deal with large structures, some details of which may easily be overlooked, such support is critical to avoid making mistakes. For example, all of the automata-based languages listed above are at least partially supported by a model-checker. The TLA Toolbox offers a full-fledged IDE based on Eclipse, which includes the TLC model-checker and the TLAPS [23] interactive proof assistant.
- Some languages have rich built-in features, like the composition operators of BIP, others are more frugal and let their users define the needed features using more primitive ones, like in TLA+.

Choosing a modeling language among the multitude that is available is a daunting task. We identify below the most important features that we need and, without doing a comparative study of the alternatives, we settle on a solution that provides us will all those features. This solution involves both the I/O-automata and TLA+ languages, and uses Isabelle/HOL as proof assistant.

There are two major constraints that guided our choice of a modeling language for this thesis:

1. We would like to represent *concisely* our objects of discussion, namely systems built out of several components which may interact by performing a *discrete joint action* and otherwise evolve completely *asynchronously*.

2. We would like *mechanically-checked proofs* to be as easy as possible.

Let us immediately observe that I/O automata are well suited to concisely represent distributed algorithms. Indeed, composing I/O automata with the I/O automaton composition operator results exactly in a system in which components, which are otherwise completely asynchronous, interact through discrete joint actions. I/O automata composition is simple and accurately models the interaction between components of a distributed system.

In contrast, taking just two examples, we don't need all the composition operators of BIP but they add some complexity to the language; in TLA+, there is no built-in notion of composition and we would have to define it manually, adding overhead to our specifications. For example, see the difference between our statement of the idempotence property of speculative linearizability using I/O automata (theorem 5.4.3) and the same statement in TLA+ (appendix A, module SpecLinCorrectness).

The second point, mechanically-checked proofs, requires a deeper examination.

8.3.1 Requirements for Tractable Mechanically-Checked Proofs

Mechanically checked proofs are extremely time-consuming and it is therefore crucial to identify the methods and tools that make them as easy as possible. Our experience in writing such proofs leads us to the following observations.

1. Fast *prototyping and debugging* tools are essential. The work presented in this thesis led to many failed proof attempts, because the statement to prove was incorrect or because we simply gave up for lack of time, but also to successful but long and tedious proofs of facts that are of minor importance. However, all those unproductive attempts sum up to several months of work. One of the problems is that the automated debugging tools available in Isabelle/HOL are not powerful enough to check high level properties of our specification. Checking low level reasoning steps was not sufficient, and we have often found out after much work that a proof was useless because some of its assumptions were incorrect. For example, thorough model-checking attempts suggest that the specification used in the composition theorem of the ALM entry of the Archive of Formal Proofs [38], whose proof took several moths to complete, is not a sound abstraction of the algorithms we intended to apply it to. We would therefore strongly advise against using mechanical theorem proving before obtained strong evidence that the conjectures to be proved are true and useful in the bigger picture.

2. Gradual *abstraction refinement* is of utmost importance, because it allows decomposing complex proofs into several simpler steps. To support abstraction refinement, a mod-

eling language must support *hiding* internal parts of a specification and must have a notion of implementation that is invariant under *stuttering*. This allows an abstract action to be implemented by a series of more concrete actions, enabling different levels of abstraction. Leslie Lamport discusses those two aspects in the context of TLA+ [57].

3. *Reduction theorems* are proved once and for all and then simplify all subsequent proofs by reducing complex statements to simpler ones. A reduction theorem allows one to reason about a simpler artifact and draw conclusions about a more complex one. An example would be the ability to reason about the components of a system one by one and draw some conclusions about the whole system (like in Speculative Linearizability), or to reason about the individual transitions of a transition system and draw conclusions about its executions (like in simulation proofs).

4. Partial *proof automation* only moderately speeds up the proof process.

The I/O automata framework is a good choice concerning points 1, 2, and 4. Indeed, I/O automata support abstraction refinement and the corresponding theory is well understood [70]. Moreover, a first reduction theorem, the monotonicity of composition with respect to the implementation relation, allows a form of compositional proofs, and the different soundness and completeness theorems relative to simulation relations allow reducing reasoning about entire traces to reasoning about individual steps and can readily be used to prove abstraction refinement. Moreover, the safety part of the theory of I/O automata is simple enough to be implemented in a few hundred lines in the interactive proof assistant Isabelle/HOL. Therefore we can benefit from the Isabelle/HOL infrastructure for partial proof automation and structured, readable proof text [102]. Proof automation in Isabelle/HOL is provided by the different built-in automatic proof methods (the simplifier, the tableau prover, the Metis prover [47], etc.) and by the external automatic provers, including SMT (Satisfiability Modulo Theories) solvers, available through Sledgehammer [9].

However, the problem with I/O automata lies in the associated prototyping and debugging tools. The only tools known to the author are a model-checker for analyzing I/O automata specification written in Isabelle/HOLCF [83, 82], developed by Müller and Nipkow, and a set of tools including a model-checker, a simulator, and a theorem prover, developed by Garland and Lynch [35, 49]. However both seem to be no longer supported. We could also use the Nitpick tool [10] of Isabelle/HOL, however our experience has shown that only modest goals can be "nitpicked".

Fortunately there is another language that has very good support for fast prototyping and debugging: it is TLA+, with the TLA Toolbox. The TLA Toolbox has several features that make it a platform of choice for fast prototyping and debugging: The TLA Toolbox offers a modern GUI to write specifications, model-check their properties with TLC, and write mechanically-checked proofs; the TLA+ language is very expressive, allowing fast prototyping of high-level designs; the TLC model-checker is able to analyze any finite state TLA+ specification almost without modifications; the TLC model-checker displays error traces graphically, making it

easy to spot errors; finally, at the time of writing, the TLA Toolbox is actively maintained and developed and has a helpful community.

However, TLA+ has drawbacks in terms of proof automation: TLAPS, the interactive theorem proving tool of the TLA+ Toolbox, is till under development and we found the encoding of TLA+ in Isabelle/HOL of Merz [37] hard to approach. Note that is most likely due to the lack of expertise with Isabelle/HOL of the author of this thesis, rather than to the Isabelle/HOL theory of Merz, and to the complexity of the theory behind TLA+.

To summarize, the I/O automata framework is a good choice for points 1,2, and 4 (abstraction refinement, proof automation, and reduction theorems), and TLA+ fills the gap by providing user-friendly prototyping and debugging tools.

Therefore, we have used the following method for developing a mechanically-checked theory of adaptive systems. We first prototype and debug our ideas using the TLA+ Toolbox. Once we are confident that our TLA+ specifications are meaningful and correct we translate them to I/O automata specifications written in Isabelle/HOL, where we carry-out the proofs.

9 Conclusion

9.1 Future Work

9.1.1 Byzantine Faults in the Speculative Linearizability Framework

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 the knowledge that processes have of 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 module 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 module 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: the interface of a mode instance would need to be augmented with actions modeling Byzantine processes acquiring knowledge about cryptographic keys and signed messages and modular properties 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 by Lynch for modeling shared key communication systems using I/O automata [68], but remains to be explored.

9.1.2 Debugging Byzantine Fault-Tolerant Algorithms

As we have observed in section 8.3.1, a mechanically-checked proofs 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, allowing to quickly explore 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. As observed by Lamport [54], TLC was no useful to check nontrivial properties of his BFT version of Paxos.

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.

9.1.3 A Proving Infrastructure in Isabelle/HOL

9.1.4 Practical Applications of Speculative Linearizability in Shared-Memory

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] Daphna Amit et al. “Comparison Under Abstraction for Verifying Linearizability”. In: *CAV*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. LNCS. Springer, 2007, pp. 477–490. DOI: 10.1007/978-3-540-73368-3_49.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Egon Börger and Robert F Stärk. *Abstract state machines: a method for high-level system design and analysis*. Vol. 14. Springer Heidelberg, 2003.

[12] 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.

[13] 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.

[14] Miguel Castro and Barbara Liskov. *A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm*. Technical Memo MIT-LCS-TM-590. MIT, 1999.

[15] 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.

[16] 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.

[17] Bernadette Charron-Bost and Stephan Merz. “Formal Verification of a Consensus Algorithm in the Heard-Of Model”. In: *Int. J. Software and Informatics* 3.2-3 (2009), pp. 273–303.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] RL Constable et al. *Implementing mathematics*. Citeseer, 1986.

[23] 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.

[24] 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.

[25] 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.

[26] Henri Debrat and Stephan Merz. “Verifying Fault-Tolerant Distributed Algorithms in the Heard-Of Model”. In: *Archive of Formal Proofs* 2012 (2012).

[27] Rainer Drexler et al. “The KIV System: A Tool for Formal Program Development”. In: *STACS*. Ed. by Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner. Vol. 665. LNCS. Springer, 1993, pp. 704–705. DOI: 10.1007/3-540-56503-5_69.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Eli Gafni and Leslie Lamport. “Disk Paxos”. In: *Distributed Computing* 16.1 (2003), pp. 1–20. DOI: 10.1007/s00446-002-0070-8.

[34] 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.

[35] Stephen J. Garland and Nancy A. Lynch. “Foundations of Component-Based Systems”. In: Cambridge University Press, 2000. Chap. Using I/O automata for developing distributed systems.

[36] 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.

[37] Gudmund Grov and Stephan Merz. “A Definitional Encoding of TLA* in Isabelle/HOL”. In: *Archive of Formal Proofs* (2011). http://afp.sf.net/entries/TLA.shtml, Formal proof development. ISSN: 2150-914x.

[38] 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.

[39] 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.

[40] Rachid Guerraoui and Marko Vukolic. “Refined quorum systems”. In: *Distributed Computing* 23.1 (2010), pp. 1–42. DOI: 10.1007/s00446-010-0103-7.

[41] 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.

[42] Maurice Herlihy. “Wait-Free Synchronization”. In: *ACM Trans. Program. Lang. Syst.* 13.1 (1991), pp. 124–149. DOI: 10.1145/114005.102808.

[43] 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.

[44] 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.

[45] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Commun. ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585.

[46] 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.

[47] Joe Hurd. “First-Order Proof Tactics in Higher-Order Logic Theorem Provers”. In: *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*. Ed. by Myla Archer, Ben Di Vito, and César Muñoz. NASA Technical Reports NASA/CP-2003-212448. 2003, pp. 56–68. URL: http://www.gilith.com/research/papers.

[48] 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.

[49] *IOA Language and Toolset (web page)*. https://groups.csail.mit.edu/tds/ioa/. Accessed: 2013-10-18. 2003.

[50] 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.

[51] 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.

[52] 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.

[53] Ramakrishna Kotla et al. “Zyzzyva: Speculative Byzantine fault tolerance”. In: *ACM Trans. Comput. Syst.* 27.4 (2009). DOI: 10.1145/1658357.1658358.

[54] 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.

[55] Leslie Lamport. “Fast Paxos”. In: *Distributed Computing* 19.2 (2006), pp. 79–103. DOI: 10.1007/s00446-006-0005-x.

[56] Leslie Lamport. *Generalized Consensus and Paxos*. https://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#generalized. Accessed: 2013-10-18. 2005.

[57] Leslie Lamport. “Logics of Specification Languages”. In: ed. by D. Bjorner and M.C. Henson. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2010. Chap. Leslie Lamport: The Specification Language TLA+.

[58] Leslie Lamport. “Lower bounds for asynchronous consensus”. In: *Distributed Computing* 19.2 (2006), pp. 104–125. DOI: 10.1007/s00446-006-0155-x.

[59] Leslie Lamport. *Mechanically Checked Safety Proof of a Byzantine Paxos Algorithm*. https://research.microsoft.com/en-us/um/people/lamport/tla/byzpaxos.html. Accessed: 2013-18-11. 2013.

[60] Leslie Lamport. “On Interprocess Communication. Part I: Basic Formalism”. In: *Distributed Computing* 1.2 (1986), pp. 77–85. DOI: 10.1007/BF01786227.

[61] Leslie Lamport. “On Interprocess Communication. Part II: Algorithms”. In: *Distributed Computing* 1.2 (1986), pp. 86–101. DOI: 10.1007/BF01786228.

[62] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X.

[63] 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.

[64] 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.

[65] Leslie Lamport and Mike Massa. “Cheap Paxos”. In: *DSN*. IEEE Computer Society, 2004, pp. 307–314. DOI: 10.1109/DSN.2004.1311900.

[66] 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.

[67] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN: 1-55860-348-4.

[68] 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.

[69] Nancy A. Lynch and Mark R. Tuttle. “An introduction to input/output automata”. In: *CWI Quarterly* 2 (1989), pp. 219–246.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] Philip K. McKinley et al. “Composing Adaptive Software”. In: *IEEE Computer* 37.7 (2004), pp. 56–64. DOI: 10.1109/MC.2004.48.

[77] Stephan Merz. “The specification language TLA+”. In: *Logics of specification languages*. Springer, 2008, pp. 401–451.

[78] 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.

[79] Robin Milner, Joachim Parrow, and David Walker. “A Calculus of Mobile Processes, I”. In: *Inf. Comput.* 100.1 (1992), pp. 1–40.

[80] Robin Milner, Joachim Parrow, and David Walker. “A Calculus of Mobile Processes, II”. In: *Inf. Comput.* 100.1 (1992), pp. 41–77.

[81] 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.

[82] Olaf Müller. “I/O Automata and Beyond: Temporal Logic and Abstraction in Isabelle”. In: *TPHOLs*. 1998, pp. 331–348.

[83] 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.

[84] 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.

[85] Peter W. O’Hearn et al. “Verifying linearizability with hindsight”. In: *PODC*. Ed. by Andréa W. Richa and Rachid Guerraoui. ACM, 2010, pp. 85–94. DOI: 10.1145/1835698.1835722.

[86] 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.

[87] 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.

[88] Lawrence C. Paulson. “A Generic Tableau Prover and its Integration with Isabelle”. In: *J. UCS* 5.3 (1999), pp. 73–87.

[89] Lawrence C. Paulson. “Isabelle: The Next 700 Theorem Provers”. In: *CoRR* cs.LO/9301106 (1993).

[90] Fernando Pedone. “Boosting System Performance with Optimistic Distributed Protocols”. In: *IEEE Computer* 34.12 (2001), pp. 80–86. DOI: 10.1109/2.970581.

[91] C. A. Petri. “Fundamentals of a Theory of Asynchronous Information Flow”. In: *IFIP Congress*. 1962, pp. 386–390.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] Gerhard Schellhorn, Heike Wehrheim, and John Derrick. “How to Prove Algorithms Linearisable”. In: *CAV*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. LNCS. Springer, 2012, pp. 243–259. DOI: 10.1007/978-3-642-31424-7_21.

[97] 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.

[98] Atul Singh et al. “BFT Protocols Under Fire”. In: *NSDI*. Ed. by Jon Crowcroft and Michael Dahlin. USENIX Association, 2008, pp. 189–204.

[99] 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.

[100] Viktor Vafeiadis. “Automatically Proving Linearizability”. In: *CAV*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Vol. 6174. LNCS. Springer, 2010, pp. 450–464. DOI: 10.1007/978-3-642-14295-6_40.

[101] Wei Wei et al. “Fast Mencius: Mencius with low commit latency”. In: *INFOCOM*. IEEE, 2013, pp. 881–889. DOI: 10.1109/INFCOM.2013.6566876.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

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*:

$$
\begin{array}{l}
TypeOk \triangleq \\
\quad \land \ \forall\, s \in S,\, c \in Req : s \bullet c \in S \\
\quad \land \ \forall\, s \in S,\, c \in Req : Output(s,\, c) \in O
\end{array}
$$

ASSUME *TypeOk*

Execute a sequence of requests:

RECURSIVE $Star(\_, \_, \_)$

$$
\begin{array}{l}
Star(s,\, rs,\, i) \triangleq \\
\quad \text{IF } Len(rs) < i \text{ THEN } s \\
\quad \text{ELSE LET } s2 \triangleq s \bullet rs[i] \text{ IN } Star(s2,\, rs,\, i+1)
\end{array}
$$

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:

$$
\begin{array}{l}
Idem1 \triangleq \forall\, s \in S : \forall\, r \in Req : \forall\, rs \in Seq(Req) : r \in Image(rs) \Rightarrow s \star rs = s \star Append(rs,\, r) \\
Idem2 \triangleq \forall\, s \in S : \forall\, o \in O : \forall\, p,\, q \in P : \forall\, c1,\, c2 \in C : \\
\quad \text{LET } r1 \triangleq \langle p,\, c1 \rangle \\
\qquad\ r2 \triangleq \langle q,\, c2 \rangle \\
\quad \text{IN} \\
\qquad Output(s,\, r1) = o \land p \neq q \\
\qquad\quad \Rightarrow \text{ LET } s2 \triangleq (s \bullet r1) \bullet r2 \\
\qquad\qquad \text{IN } \ Output(s2,\, r1) = o \\
Idem \triangleq Idem1 \land Idem2
\end{array}
$$

ASSUME *Idem*

The partial order:

$$
\begin{array}{l}
PrecEq(s1,\, s2) \triangleq \\
\quad \lor \ s1 = s2 \\
\quad \lor \ \exists\, rs \in Seq(Req) : s2 = s1 \star rs
\end{array}
$$

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:

$IsLB(s,\, s1,\, s2) \triangleq s \preceq s1 \land s \preceq s2$

$IsGLB(s,\, s1,\, s2) \triangleq$

$\quad \land\, IsLB(s,\, s1,\, s2)$

$\quad \land \forall\, s3 \in S : s \neq s3 \land IsLB(s3,\, s1,\, s2) \Rightarrow s3 \preceq s$

Semi lattice property of *RDRs*:

$s1 \sqcup s2 \triangleq$ CHOOSE $s \in S : IsGLB(s,\, s1,\, s2)$

$GLBExists \triangleq \forall\, s1,\, s2 \in S : IsGLB(s1 \sqcup s2,\, s1,\, s2)$

ASSUME $GLBExists$

GLB of a set of states:

RECURSIVE $GLB1(\_)$

$GLB1(ss) \triangleq$

```
    LET s ≜ CHOOSE s ∈ ss : TRUE
    IN
        IF Cardinality(ss) = 1
        THEN s
        ELSE (s ⊔ GLB1(ss \ {s}))
```

ASSUME $\forall\, ss \in$ SUBSET $S : GLB1(ss) = GLB(ss)$

The consistency property of *RDRs*:

$Consistency \triangleq \forall\, s0,\, s1,\, s2 \in S,\, rs1,\, rs2 \in Seq(Req) :$

```
    LET rset ≜ Image(rs1) ∪ Image(rs2)
    IN  ∧ s1 = s0 ⋆ rs1
        ∧ s2 = s0 ⋆ rs2
        ⇒ ∃ rs ∈ Seq(rset) : s1 ⊔ s2 = s0 ⋆ 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 ≜ {"ts"}
O ≜ {"Won", "Lost"}
S ≜ {P} ∪ P
Bot ≜ P
s • r ≜
 IF s = P THEN r[1] ELSE s
Output(s, r) ≜
  IF s = P THEN "Won" ELSE IF r[1] = s THEN "Won" ELSE "Lost"
s1 ≼ s2 ≜
   ∨ s1 = s2 ∨ s1 = P
s ⋆ rs ≜
 IF rs = ⟨⟩ THEN s
  ELSE rs[1][1]
GLB(ss) ≜
  IF ss = {} THEN ⟨⟩
   ELSE
    IF ∃ s1, s2 ∈ ss : s1 ≠ s2
     THEN P
     ELSE CHOOSE s ∈ ss : TRUE
Contains(s, r) ≜
  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 $rs = \langle\rangle \lor s \neq V$ THEN s
ELSE $rs[1][2]$

$GLB(ss) \triangleq$
IF $ss = \{\}$ THEN $\langle\rangle$
ELSE
IF $\exists\, s1,\ s2 \in ss : s1 \neq s2$
THEN V
ELSE CHOOSE $s \in ss$: TRUE

$Contains(s,\ r) \triangleq$
IF $s = V$ THEN FALSE ELSE TRUE

MODULE *Generic*

EXTENDS *Library*

CONSTANTS P, C
$O \triangleq Seq(P \times C)$
$S \triangleq \{rs \in Seq(P \times C) : NoDup(rs, \{\})\}$
$Bot \triangleq \langle\rangle$
$s \bullet r \triangleq$ IF $r \in Image(s)$ THEN s ELSE $Append(s, r)$
$Output(s, r) \triangleq$ IF $r \in Image(s)$ THEN $Truncate(r, s)$ ELSE $Append(s, r)$
$s1 \preceq s2 \triangleq$
  $Prefix(s1, s2)$
$s \star rs \triangleq s \circ RemDup(rs)$
$GLB(ss) \triangleq LongestCommonPrefix(ss)$
$Contains(s, r) \triangleq r \in Image(s)$

MODULE *LinInterface*

EXTENDS *Library*

CONSTANTS P, C, S, O
VARIABLE *interface*

$InvInterfaceType \triangleq [P \to [cmd : C, flag : \text{BOOLEAN }]]$
$RespInterfaceType \triangleq [P \to [output : O, flag : \text{BOOLEAN }]]$

$InterfaceType \triangleq [$
  $inv : InvInterfaceType,$
  $resp : RespInterfaceType]$

$InvInterfaceInit \triangleq [p \in P \mapsto [$
  $cmd \mapsto Some(C),$
  $flag \mapsto Some(\text{BOOLEAN })]]$

$RespInterfaceInit \triangleq [p \in P \mapsto [$
  $output \mapsto Some(O),$
  $flag \mapsto Some(\text{BOOLEAN })]]$

$InterfaceInit \triangleq [$
  $inv \mapsto InvInterfaceInit,$
  $resp \mapsto RespInterfaceInit]$

$Invoke(p, cmd) \triangleq$
  $interface' = [interface \text{ EXCEPT } !.inv = [@ \text{ EXCEPT } ![p] = [$
    $cmd \mapsto cmd,$
    $flag \mapsto \neg@.flag]]]$

$Response(p, o) \triangleq$
  $interface' = [interface \text{ EXCEPT } !.resp = [@ \text{ EXCEPT } ![p] = [$
    $output \mapsto o,$
    $flag \mapsto \neg@.flag]]]$

MODULE *Linearizability*

EXTENDS *RDR*

VARIABLES
status, *pending*, *dState*, *nxtOut*, *interface*

INSTANCE *LinInterface*

$vars \triangleq \langle status, pending, dState, nxtOut, interface \rangle$

$Label \triangleq \{\text{“ready”}, \text{“committed”}, \text{“pending”}\}$ The status of a process.

$TypeInvariant \triangleq$
$\forall\, p \in P :$
$\land\ status[p] \in Label$
$\land\ pending[p] \in C$
$\land\ nxtOut[p] = O$
$\land\ dState \in S$

Invocation by process p:
$Inv(p) \triangleq \exists\, c \in C :$
$\land\ status[p] = \text{“ready”}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“pending”}]$
$\land\ pending' = [pending \text{ EXCEPT } ![p] = \langle p, c \rangle]$
$\land\ Invoke(p, c)$
$\land$ UNCHANGED $\langle dState, nxtOut \rangle$

Response by process p:
$Resp(p) \triangleq$
$\land\ status[p] = \text{“committed”}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“ready”}]$
$\land\ Response(p, nxtOut[p])$
$\land$ UNCHANGED $\langle dState, pending, nxtOut \rangle$

Linearize one pending request.
$Lin \triangleq$
$\land\, \exists\, p \in P :$
$\land\ status[p] = \text{“pending”}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“committed”}]$
$\land\ dState' = dState \bullet pending[p]$
$\land\ nxtOut' = [nxtOut \text{ EXCEPT } ![p] = Output(dState, pending[p])]$
$\land$ UNCHANGED $\langle pending, interface \rangle$

$Init \triangleq$
$\land\ status = [p \in P \mapsto \text{“ready”}]$
$\land\ dState = Bot$
$\land\ pending = [p \in P \mapsto Some(Req)]$

$\land\ nxtOut = [p \in P \mapsto Some(O)]$
$\land\ interface = InterfaceInit$

$Next \triangleq Lin \lor (\exists\, p \in P : Inv(p) \lor Resp(p))$

$Spec \triangleq Init \land \Box[Next]_{vars}$

MODULE *SpecLinInterface*

EXTENDS *Library*

CONSTANTS P, C, S, O

VARIABLE *interface*

$LI \triangleq$ INSTANCE *LinInterface*

$SwitchInterfaceType \triangleq [P \to [cmd : C,\ sval : S,\ flag : \text{BOOLEAN}\ ]]$

$InterfaceType \triangleq [$
  $init :\ SwitchInterfaceType,$
  $inv : LI!InvInterfaceType,$
  $resp : LI!RespInterfaceType,$
  $abort : SwitchInterfaceType]$

$SwitchInterfaceInit \triangleq [p \in P \mapsto [$
  $cmd \mapsto Some(C),$
  $sval \mapsto Some(S),$
  $flag \mapsto Some(\text{BOOLEAN}\ )]]$

$InterfaceInit \triangleq [$
  $init \mapsto SwitchInterfaceInit,$
  $inv \mapsto LI!InvInterfaceInit,$
  $resp \mapsto LI!RespInterfaceInit,$
  $abort \mapsto SwitchInterfaceInit]$

$Invoke(p,\ cmd) \triangleq LI!Invoke(p,\ cmd)$

$Response(p,\ o) \triangleq LI!Response(p,\ o)$

$Initialize(p,\ cmd,\ sv) \triangleq$
  $interface' = [interface\ \text{EXCEPT}\ !.init = [@\ \text{EXCEPT}\ ![p] = [$
    $cmd \mapsto cmd,$
    $sval \mapsto sv,$
    $flag \mapsto \neg@.flag]]]$

$Abort(p,\ cmd,\ sv) \triangleq$
  $interface' = [interface\ \text{EXCEPT}\ !.abort = [@\ \text{EXCEPT}\ ![p] = [$
    $cmd \mapsto cmd,$
    $sval \mapsto sv,$
    $flag \mapsto \neg@.flag]]]$

MODULE *SpecLin*

EXTENDS *Library*, *RDR*

CONSTANT *Initial* TRUE when first instance.

VARIABLES
status, *pending*, *dState*, *initialized*, *abortVals*, *initVals*, *interface*

INSTANCE *SpecLinInterface*

$vars \triangleq \langle status, pending, dState, interface, initVals, initialized, abortVals \rangle$
$statusStr \triangleq \{\text{“idle”}, \text{“ready”}, \text{“aborted”}, \text{“pending”}\}$

$TypeInvariant \triangleq$
$\land \forall\, p \in P :$
$\quad \land\ status[p] \in statusStr$
$\quad \land\ pending[p] \in Req$
$\land\ dState \in S$
$\land\ initVals \in \text{SUBSET}\ S$
$\land\ abortVals \in \text{SUBSET}\ S$

Initial states
$Init \triangleq$
$\land$ IF $Initial$
THEN $\land\ status = [p \in P \mapsto \text{“ready”}]$
$\land\ initialized\ = \text{TRUE}$
ELSE $\land\ status = [p \in P \mapsto \text{“idle”}]$
$\land\ initialized\ = \text{FALSE}$
$\land\ dState = Bot$
$\land\ pending = [p \in P \mapsto Some(Req)]$
$\land\ initVals = \{\}$
$\land\ abortVals = \{\}$
$\land\ interface\ = InterfaceInit$

Invocation by process p:
$Inv(p) \triangleq \exists\, c \in C :$
$\land\ status[p] = \text{“ready”}$
$\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“pending”}]$
$\land\ pending' = [pending\ \text{EXCEPT}\ ![p] = \langle p, c \rangle]$
$\land\ Invoke(p, c)$
$\land\ \text{UNCHANGED}\ \langle dState, initialized, initVals, abortVals \rangle$

Response by process p:
$Resp(p) \triangleq$
$\land\ status[p] = \text{“pending”}$
$\land\ initialized$
$\land\ status' = [status\ \text{EXCEPT}\ ![p] = \text{“ready”}]$

```
    ∧ Contains(dState, pending[p])
    ∧ Response(p, Output(dState, pending[p]))
    ∧ UNCHANGED ⟨dState, pending, initialized, initVals, abortVals⟩

Pending ≜
  {p ∈ P : status[p] ∈ {"pending", "aborted"}}
PendingReqs ≜
  {pending[p] : p ∈ Pending}
InitSets ≜
  {is ∈ SUBSET initVals : is ≠ {}}

SafeInit ≜
  {s1 ∈ S :
    ∧ initVals ≠ {}
    ∧ ∃ is ∈ InitSets :
      ∃ rs ∈ NoDupSeq1(PendingReqs) :
        s1 = GLB(is) ⋆ rs
    ∧ ∀ a ∈ abortVals : s1 ⪯ a}

SafeCommit ≜
  {s1 ∈ S :
    ∧ dState ⪯ s1
    ∧ ∨ ∃ rs ∈ NoDupSeq1(PendingReqs) : s1 = dState ⋆ rs
      ∨ ∃ is ∈ InitSets :
        ∧ dState ⪯ GLB(is)
        ∧ ∃ rs ∈ NoDupSeq1(PendingReqs) : s1 = GLB(is) ⋆ rs}

SafeAbort ≜
  {s1 ∈ S :
    IF initialized
    THEN s1 ∈ SafeCommit
    ELSE ∃ is ∈ InitSets :
      ∃ rs ∈ NoDupSeq1(PendingReqs) :
        s1 = GLB(is) ⋆ rs}

 Abort by process p:
Abo(p) ≜
  ∧ status[p] = "pending"
  ∧ status' = [status EXCEPT ![p] = "aborted"]
  ∧ ∃ s1 ∈ SafeAbort :
    ∧ Abort(p, pending[p][2], s1)
    ∧ abortVals' = abortVals ∪ {s1}
  ∧ UNCHANGED ⟨dState, pending, initialized, initVals⟩

 Linearize some pending requests.
Lin ≜
    ∧ initialized
```

$\land\ PendingReqs \neq \{\}$
$\land\ \exists\, s \in SafeCommit :$
$\quad \land\ \forall\, av \in abortVals : s \preceq av$
$\quad \land\ dState' = s$
$\land\ dState' \in S$ For TLC
$\land$ UNCHANGED $\langle status,\ pending,\ interface,\ initialized,\ initVals,\ abortVals\rangle$

Init call
$Ini(p) \triangleq$
$\land\ status[p] =$ "idle"
$\land\ \exists\, c \in C,\ sval \in S :$
$\quad \land\ Initialize(p,\ c,\ sval)$
$\quad \land\ status' = [status$ EXCEPT $![p] =$ "pending"$]$
$\quad \land\ pending' = [pending$ EXCEPT $![p] = \langle p,\ c\rangle]$
$\quad \land\ initVals' = initVals \cup \{sval\}$
$\land$ 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*

$Mode1(status, pending, dState, initialized, initVals, abortVals) \triangleq$
INSTANCE *SpecLin* WITH *Initial* $\leftarrow$ TRUE

$Lin(status, pending, dState, nxtOut) \triangleq$ INSTANCE *Linearizability* WITH
interface $\leftarrow [inv \mapsto interface.inv, resp \mapsto interface.resp]$

$Mode1Spec \triangleq \exists\, status, pending, s, initVals, abortVals, initialized :$
$Mode1(status, pending, s, initVals, abortVals, initialized)!Spec$

$LinSpec \triangleq \exists\, status, pending, s, nxtOut : Lin(status, pending, s, nxtOut)!Spec$

THEOREM $Mode1Spec \Rightarrow LinSpec$

MODULE *SpecLinIsIdemPotent*

Here we compose two instances of speculative linearizability using the "explicit state changes" method (see Specifying Systems, page 144-147). A joint-action specification would complicate the refinement because, for example, two requests could be linearized at the same time in the two different instances.

EXTENDS *SpecLinInterface*

$SingleMode(status, pending, dState, initVals, abortVals, initialized) \triangleq$
INSTANCE *SpecLin* WITH *Initial* $\leftarrow$ TRUE

MODULE *Composition*

VARIABLES $status1, pending1, dState1, initVals1, abortVals1, initialized1, interface1$
$vars1 \triangleq \langle status1, pending1, dState1, initVals1, abortVals1, initialized1,$
$interface1\rangle$
$Mode1 \triangleq$ INSTANCE *SpecLin* WITH
Initial $\leftarrow$ TRUE,
$status \leftarrow status1$, $pending \leftarrow pending1$, $dState \leftarrow dState1$, $initVals \leftarrow initVals1$,
$abortVals \leftarrow abortVals1$,
$initialized \leftarrow initialized1$, $interface \leftarrow interface1$

VARIABLES $status2, pending2, dState2, initVals2, abortVals2, initialized2, interface2$
$vars2 \triangleq \langle status2, pending2, dState2, initVals2, abortVals2, initialized2,$
$interface2\rangle$
$Mode2 \triangleq$ INSTANCE *SpecLin* WITH
Initial $\leftarrow$ FALSE,

$status \leftarrow status2,\ pending \leftarrow pending2,\ dState \leftarrow dState2,\ initVals \leftarrow initVals2,$
$abortVals \leftarrow abortVals2,$
$initialized \leftarrow initialized2,\ interface \leftarrow interface2$

$$
\begin{array}{l}
LinkInterfaces \triangleq \\
\quad \land\ interface1'.abort = interface2'.init \\
\quad \land\, \forall\, p \in P : \\
\qquad \land\ interface1'.inv[p] \neq interface1.inv[p] \\
\qquad\quad \Rightarrow interface.inv' = [interface.inv \text{ EXCEPT } ![p] \\
\qquad\qquad\quad = [@ \text{ EXCEPT } !.cmd = interface1'.inv[p].cmd, \\
\qquad\qquad\qquad\qquad !.flag = \neg @]] \\
\qquad \land\ interface1'.resp[p] \neq interface1.resp[p] \\
\qquad\quad \Rightarrow interface.resp' = [interface.resp \text{ EXCEPT } ![p] \\
\qquad\qquad\quad = [@ \text{ EXCEPT } !.output = interface1'.resp[p].output, \\
\qquad\qquad\qquad\qquad !.flag = \neg @]] \\
\qquad \land\ interface2'.inv[p] \neq interface2.inv[p] \\
\qquad\quad \Rightarrow interface.inv' = [interface.inv \text{ EXCEPT } ![p] \\
\qquad\qquad\quad = [@ \text{ EXCEPT } !.cmd = interface2'.inv[p].cmd, \\
\qquad\qquad\qquad\qquad !.flag = \neg @]] \\
\qquad \land\ interface2'.resp[p] \neq interface2.resp[p] \\
\qquad\quad \Rightarrow interface.resp' = [interface.resp \text{ EXCEPT } ![p] \\
\qquad\qquad\quad = [@ \text{ EXCEPT } !.output = interface2'.resp[p].output, \\
\qquad\qquad\qquad\qquad !.flag = \neg @]] \\
\quad \land\ interface1'.interface.inv = interface1.interface.inv \\
\quad \land\ interface2'.interface.inv = interface2.interface.inv \\
\qquad \Rightarrow interface.inv' = interface.inv \\
\quad \land\ interface1'.interface.resp = interface1.interface.resp \\
\quad \land\ interface2'.interface.resp = interface2.interface.resp \\
\qquad \Rightarrow interface.resp' = interface.resp \\
\quad \land\ interface1'.abort = interface2'.init
\end{array}
$$

$$
\begin{array}{l}
CompoNext \triangleq \\
\quad \land\ \lor\ \land\ Mode1!Next \\
\qquad\qquad\ \land\ \text{UNCHANGED } vars2 \\
\qquad\quad \lor\ \land\ Mode2!Next \\
\qquad\qquad\ \land\ \text{UNCHANGED } vars1 \\
\qquad\quad \lor \exists\, p \in P : Mode1!Abo(p) \land Mode2!Ini(p) \\
\quad \land\ LinkInterfaces
\end{array}
$$

$$CompoInit \triangleq Mode1!Init \land Mode2!Init \land interface = InterfaceInit$$

$$CompoSpec \triangleq CompoInit \land \Box[CompoNext]_{\langle vars1,\, vars2,\, interface\rangle}$$

$Compo(status1,\ pending1,\ dState1,\ initVals1,\ abortVals1,\ initialized1,\ interface1,$
$status2,\ pending2,\ dState2,\ initVals2,\ abortVals2,\ initialized2,\ interface2)$

$\triangleq$ INSTANCE *Composition*

CompoSpec $\triangleq$
$\exists$ *status*1, *pending*1, *dState*1, *initVals*1, *abortVals*1, *initialized*1, *interface*1 :
$\exists$ *status*2, *pending*2, *dState*2, *initVals*2, *abortVals*2, *initialized*2, *interface*2 :
Compo(*status*1, *pending*1, *dState*1, *initVals*1, *abortVals*1, *initialized*1,
*interface*1, *status*2, *pending*2, *dState*2, *initVals*2, *abortVals*2,
*initialized*2, *interface*2)!*CompoSpec*

SingleModeSpec $\triangleq$
$\exists$ *status*, *pending*, *dState*, *initVals*, *abortVals*, *initialized* :
SingleMode(*status*, *pending*, *dState*, *initVals*, *abortVals*, *initialized*)!*Spec*

THEOREM *CompoSpec* $\Rightarrow$ *SingleModeSpec*

A.2 Message-Passing Adaptive Algorithms

MODULE *MPGC*

EXTENDS *RDR*, *Library*

CONSTANTS *Initial*, *Acceptor*

RespQuorum is the set of quorums used to determine a response
AbortQuorum is the set of quorums used to determine an abort value
AbortValues([$Q \to S$]) is the set of safe abort values given
the *dStates* of a quorum Q of acceptors.

CONSTANTS *RespQuorum*, *AbortQuorum*, *AbortValues*(_)

VARIABLES *status*, *pending*, *initVals*, *dState*, *accStatus*, *interface*, *pastPending*

abortVals is a history variable

VARIABLE *abortVals*

INSTANCE *SpecLinInterface*

$vars \triangleq \langle status, pending, initVals, dState, accStatus, interface, abortVals, pastPending\rangle$
$Labels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“pending”}, \text{“panic”}, \text{“aborted”}\}$
$AcceptorLabels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“stopped”}\}$

$TypeInvariant \triangleq$
$\land \forall\, p \in P :$
$\quad \land\ status[p] \in Labels$
$\quad \land\ pending[p] \in Req$
$\land \forall\, r \in Acceptor :$
$\quad \land\ dState[r] \in S$
$\quad \land\ accStatus[r] \in AcceptorLabels$
$\land\ initVals = \{\}$
$\land\ pastPending \subseteq Req$

$Init \triangleq$
$\land\ status =$
$\quad$ IF $Initial$
$\quad\quad$ THEN $[p \in P \mapsto \text{“ready”}]$
$\quad\quad$ ELSE $[p \in P \mapsto \text{“idle”}]$
$\land\ pending = [p \in P \mapsto Some(Req)]$
$\land\ initVals = \{\}$
$\land\ dState = [r \in Acceptor \mapsto Bot]$
$\land\ accStatus =$
$\quad$ IF $Initial$
$\quad\quad$ THEN $[r \in Acceptor \mapsto \text{“ready”}]$
$\quad\quad$ ELSE $[r \in Acceptor \mapsto \text{“idle”}]$
$\land\ interface = InterfaceInit$

$\land\ abortVals = \{\}$
$\land\ pastPending = \{\}$

$Ini(p) \triangleq \exists\, c \in C,\ v \in S :$
$\land\ status[p] = \text{“idle”}$
$\land\ pending' = [pending \text{ EXCEPT } ![p] = \langle p,\ c\rangle]$
$\land\ initVals' = initVals \cup \{v\}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“pending”}]$
$\land\ Initialize(p,\ c,\ v)$
$\land\ pastPending' = pastPending \cup \{\langle p,\ c\rangle\}$
$\land\ \text{UNCHANGED } \langle dState,\ accStatus,\ abortVals\rangle$

$Inv(p) \triangleq \exists\, c \in C :$
$\land\ status[p] = \text{“ready”}$
$\land\ pending' = [pending \text{ EXCEPT } ![p] = \langle p,\ c\rangle]$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“pending”}]$
$\land\ Invoke(p,\ c)$
$\land\ pastPending' = pastPending \cup \{\langle p,\ c\rangle\}$
$\land\ \text{UNCHANGED } \langle dState,\ accStatus,\ initVals,\ abortVals\rangle$

$SrvStates(Q) \triangleq$
$\{s \in S : \exists\, srv \in Q : s = dState[srv]\}$

$Res(p) \triangleq$
$\land\ status[p] = \text{“pending”}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“ready”}]$
$\land\ \exists\, Q \in RespQuorum :$
$\quad\land\ \forall\, srv \in Q : accStatus[srv] \neq \text{“idle”}$
$\quad\land\ \text{LET } glb \triangleq GLB(SrvStates(Q))$
$\qquad\text{IN}\quad\land\ Contains(glb,\ pending[p])$
$\qquad\qquad\land\ Response(p,\ Output(glb,\ pending[p]))$
$\land\ \text{UNCHANGED } \langle pending,\ initVals,\ dState,\ accStatus,$
$abortVals,\ pastPending\rangle$

Computing an abort value: all thirds contain at least one RDR of the Quorum that was used for the last commit. Therefore every GLB is either a prefix of the last committed RDR or an extension of it with pending requests.

$Abo(p) \triangleq$
$\land\ status[p] = \text{“panic”}$
$\land\ \exists\, Q \in AbortQuorum :$
$\quad\land\ \forall\, srv \in Q : accStatus[srv] = \text{“stopped”}$
$\quad\land\ \exists\, s \in AbortValues([a \in Q \mapsto dState[a]]) :$
$\qquad\land\ Abort(p,\ pending[p][2],\ s)$
$\qquad\land\ abortVals' = abortVals \cup \{s\}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“aborted”}]$
$\land\ \text{UNCHANGED } \langle pending,\ initVals,\ dState,\ accStatus,\ pastPending\rangle$

```
 We abstract over time: a process can panic at any moment.
Panic(p) ≜
  ∧  status[p] = "pending"
  ∧  status' = [status EXCEPT ![p] = "panic"]
  ∧  UNCHANGED ⟨pending, initVals, dState, accStatus, interface,
     abortVals, pastPending⟩

Pending ≜
  {p ∈ P : status[p] ∈ {"pending", "panic", "aborted"}}

 A Acceptor executes a pending request.
Exec(r) ≜
  ∧ accStatus[r] = "ready"
  ∧ ∃ req ∈ pastPending :
     dState' = [dState EXCEPT ![r] = @ • req]
  ∧ UNCHANGED ⟨status, pending, initVals, accStatus, interface,
     abortVals, pastPending⟩

 A Acceptor sets its local state to one of the init values of the processes.
WakeUp(r) ≜
  ∧ accStatus[r] = "idle"
  ∧ ∃ iv ∈ initVals :
     ∧  dState' = [dState EXCEPT ![r] = iv]
     ∧  accStatus' = [accStatus EXCEPT ![r] = "ready"]
  ∧ UNCHANGED ⟨status, pending, initVals, interface, abortVals, pastPending⟩

Stop(r) ≜ ∃ p ∈ P :
     ∧  status[p] ∈ {"panic", "aborted"}
     ∧  accStatus[r] = "ready"
     ∧  accStatus' = [accStatus EXCEPT ![r] = "stopped"]
     ∧  UNCHANGED ⟨status, pending, initVals, interface, dState,
        abortVals, pastPending⟩

Next ≜
     ∨  ∃ p ∈ P : Ini(p) ∨ Inv(p) ∨ Res(p) ∨ Abo(p) ∨ Panic(p)
     ∨  ∃ r ∈ Acceptor : Exec(r) ∨ WakeUp(r) ∨ Stop(r)

Spec ≜ Init ∧ □[Next]_vars
```

MODULE *FastMPGC*

EXTENDS *FiniteSets*, *Naturals*, *Library*, *Consensus*, *TLCDefs*

CONSTANTS *Initial*, *Acceptor*

RespQuorum is the set of quorums used to determine a response
AbortQuorum is the set of quorums used to determine an abort value
AbortValues([$Q \to S$]) is the set of safe abort values given
the *dStates* of a quorum Q of acceptors.

CONSTANTS *RespQuorum*, *AbortQuorum*

INSTANCE *FastMPGCDefs*

VARIABLES *status*, *pending*, *initVals*, *dState*, *accStatus*, *interface*,
abortVals, *pastPending*

INSTANCE *MPGC*

$slin\_status \triangleq [p \in P \mapsto \text{IF } status[p] \in \{\text{“pending”}, \text{“panic”}\} \text{ THEN “pending” ELSE } status[p]]$
$slin\_pending \triangleq pending$
$slin\_dState \triangleq Max(\{GLB(SrvStates(Q)) : Q \in RespQuorum\}, \text{LAMBDA } a, b : a \preceq b)$
$slin\_interface \triangleq interface$
$slin\_initialized \triangleq$
IF *Initial* THEN TRUE
ELSE $\exists\, Q \in RespQuorum : \forall\, a \in Q : accStatus[a] \neq$ “idle”
$slin\_initVals \triangleq initVals$
$slin\_abortVals \triangleq abortVals$

$SLin \triangleq$ INSTANCE *SpecLin* WITH
status $\leftarrow$ *slin_status*,
pending $\leftarrow$ *slin_pending*,
dState $\leftarrow$ *slin_dState*,
interface $\leftarrow$ *slin_interface*,
initialized $\leftarrow$ *slin_initialized*,
initVals $\leftarrow$ *slin_initVals*,
abortVals $\leftarrow$ *slin_abortVals*

THEOREM $Spec \Rightarrow SLin!Spec$

MODULE *SafeMPGC*

EXTENDS *FiniteSets*, *Naturals*, *Library*, *Generic*, *TLCDefs*

CONSTANTS *Initial*, *Acceptor*

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.

$$
\begin{array}{l}
AcceptorConsistency \triangleq \\
\quad \forall\, acc1,\, acc2 \in Acceptor : \\
\qquad \text{LET } s1 \triangleq dState[acc1] \\
\qquad\qquad\; s2 \triangleq dState[acc2] \\
\qquad \text{IN } \; s1 \preceq s2 \lor s2 \preceq s1
\end{array}
$$

INSTANCE *MPGC*

$$ConsistentSpec \triangleq Init \land \Box[Next \land AcceptorConsistency']_{vars}$$

$$slin\_status \triangleq [p \in P \mapsto \text{IF } status[p] \in \{\text{“pending”}, \text{“panic”}\} \text{ THEN “pending” ELSE } status[p]]$$
$$slin\_pending \triangleq pending$$
$$slin\_dState \triangleq Max(\{GLB(SrvStates(Q)) : Q \in RespQuorum\}, \text{LAMBDA } a,\, b : a \preceq b)$$
$$slin\_interface \triangleq interface$$
$$
\begin{array}{l}
slin\_initialized \triangleq \\
\quad \text{IF } Initial \text{ THEN TRUE} \\
\quad \text{ELSE } \exists\, Q \in RespQuorum : \forall\, a \in Q : accStatus[a] \neq \text{“idle”}
\end{array}
$$
$$slin\_initVals \triangleq initVals$$
$$slin\_abortVals \triangleq abortVals$$

$$
\begin{array}{l}
SLin \triangleq \text{INSTANCE } SpecLin \text{ WITH} \\
\quad status \leftarrow slin\_status, \\
\quad pending \leftarrow slin\_pending, \\
\quad dState \leftarrow slin\_dState, \\
\quad interface \leftarrow slin\_interface, \\
\quad initialized \leftarrow slin\_initialized, \\
\quad initVals \leftarrow slin\_initVals, \\
\quad abortVals \leftarrow slin\_abortVals
\end{array}
$$

THEOREM $ConsistentSpec \Rightarrow SLin!Spec$

MODULE *Quorum*

EXTENDS *Consensus*, *Library*, *TLCDefs*
INSTANCE *RDR*

CONSTANTS *AbortQuorum*, *RespQuorum*, *Initial*, *Acceptor*

INSTANCE *FastMPGCDefs*

VARIABLES *status*, *pending*, *initVals*, *execAcks*, *panicAcks*, *dState*,
network, *accStatus*, *interface*
abortVals and *pastPending* are history variables
VARIABLE *abortVals*, *pastPending*

INSTANCE *SpecLinInterface*

$vars \triangleq \langle status, pending, initVals, execAcks, panicAcks, dState, accStatus, interface, network, abortVals, pastPending\rangle$

$Labels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“pending”}, \text{“panic”}, \text{“aborted”}\}$
$AcceptorLabels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“stopped”}\}$

$Agent \triangleq P \cup Acceptor$
TLC must be able to test members of a set for equality, therefore one cannot have the following set: {1, TRUE}. Since *TLC* can test equality of sequences pointwise starting with the first element, we will *maCe* sure that messages are sequences whose first element is a string.
$Msg \triangleq \{\langle \text{“req”}, r\rangle : r \in Req\} \cup \{\langle \text{“execAck”}, s\rangle : s \in S\}$
$\cup \{\langle \text{“panic”}\rangle\} \cup \{\langle \text{“panicAck”}, s\rangle : s \in S\}$
$\cup \{\langle \text{“init”}, s\rangle : s \in S\}$
INSTANCE *Network*

$TypeInvariant \triangleq$
$\land \forall p \in P :$
$\land status[p] \in Labels$
$\land pending[p] \in Req$
$\land \forall a \in Acceptor :$
$\land execAcks[p][a] \in \{\{s\} : s \in S\} \cup \{\{\}\}$
$\land panicAcks[p][a] \in \{\{s\} : s \in S\} \cup \{\{\}\}$
$\land \forall a \in Acceptor :$
$\land dState[a] \in S$
$\land accStatus[a] \in AcceptorLabels$
$\land initVals \subseteq S$
$\land abortVals \subseteq S$
$\land pastPending \subseteq Req$

The processes

$InitProcs \triangleq$
$\land\ status = [p \in P \mapsto \text{IF } Initial \text{ THEN } \text{“ready”} \text{ ELSE } \text{“idle”}]$
$\land\ pending = [p \in P \mapsto Some(Req)]$
$\land\ execAcks = [p \in P \mapsto [a \in Acceptor \mapsto \{\}]]$
$\land\ panicAcks = [p \in P \mapsto [a \in Acceptor \mapsto \{\}]]$

$Inv(p) \triangleq$
$\land\ status[p] = \text{“ready”}$
$\land \exists\, c \in C :$
$\quad \land\ Invoke(p,\ c)$
$\quad \land\ Snd(p,\ [a \in Acceptor \mapsto \{\langle \text{“req”},\ \langle p,\ c\rangle\rangle\}])$
$\quad \land\ pending' = [pending \text{ EXCEPT } ![p] = \langle p,\ c\rangle]$
$\quad \land\ pastPending' = pastPending \cup \{\langle p,\ c\rangle\}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“pending”}]$
$\land \text{UNCHANGED } \langle initVals,\ execAcks,\ panicAcks,\ dState,\ accStatus,$
$\quad abortVals\rangle$

$Ini(p) \triangleq$
$\land\ status[p] = \text{“idle”}$
$\land \exists\, c \in C,\ s \in S :$
$\quad \land\ Initialize(p,\ c,\ s)$
$\quad \land\ Snd(p,\ [a \in Acceptor \mapsto \{\langle \text{“init”},\ s\rangle,\ \langle \text{“req”},\ \langle p,\ c\rangle\rangle\}])$
$\quad \land\ pending' = [pending \text{ EXCEPT } ![p] = \langle p,\ c\rangle]$
$\quad \land\ initVals' = initVals \cup \{s\}$
$\quad \land\ pastPending' = pastPending \cup \{\langle p,\ c\rangle\}$
$\land\ status' = [status \text{ EXCEPT } ![p] = \text{“pending”}]$
$\land \text{UNCHANGED } \langle execAcks,\ panicAcks,\ dState,\ accStatus,\ abortVals\rangle$

$RcvExecAcC(p) \triangleq$
$\land\ status[p] \in \{\text{“pending”},\ \text{“panic”}\}$
$\land \exists\, s \in S,\ a \in Acceptor :$
$\quad \land\ Rcv(p,\ \langle \text{“execAck”},\ s\rangle,\ a)$
$\quad \land\ execAcks' = [execAcks \text{ EXCEPT } ![p] = [@ \text{ EXCEPT } ![a] = \{s\}]]$
$\land \text{UNCHANGED } \langle status,\ pending,\ initVals,\ dState,\ accStatus,$
$\quad interface,\ abortVals,\ panicAcks,\ pastPending\rangle$

$RcvPanicAck(p) \triangleq$
$\land\ status[p] = \text{“panic”}$
$\land \exists\, s \in S,\ a \in Acceptor :$
$\quad \land\ Rcv(p,\ \langle \text{“panicAck”},\ s\rangle,\ a)$
$\quad \land\ panicAcks' = [panicAcks \text{ EXCEPT } ![p] = [@ \text{ EXCEPT } ![a] = \{s\}]]$
$\land \text{UNCHANGED } \langle status,\ pending,\ execAcks,\ initVals,\ dState,$
$\quad accStatus,\ interface,\ abortVals,\ pastPending\rangle$

A process can panic at any time because it times out.
$Panic(p) \triangleq$

```
  ∧  status[p] = "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 = [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]
```

$\land$ UNCHANGED $\langle status, initVals, panicAcks, pending, execAcks, interface, abortVals, pastPending\rangle$

$Exec(a) \triangleq$
$\land\ accStatus[a] = \text{“ready”}$
$\land \exists\, p \in P,\ req \in Req :$
$\land\ RcvSnd(a, \langle \text{“req”}, req\rangle, p,$
$[q \in \{p\} \mapsto \{\langle \text{“execAck”}, dState[a] \bullet req\rangle\}])$
$\land\ dState' = [dState \text{ EXCEPT } ![a] = @ \bullet req]$
$\land\ dState'[a] \in S$ For *TLC*
$\land$ UNCHANGED $\langle status, initVals, pending, execAcks, interface, accStatus, abortVals, panicAcks, pastPending\rangle$

$Stop(a) \triangleq$
$\land\ accStatus[a] = \text{“ready”}$
$\land \exists\, p \in P : RcvSnd(a, \langle \text{“panic”}\rangle, p,$
$[q \in \{p\} \mapsto \{\langle \text{“panicAck”}, dState[a]\rangle\}])$
$\land\ accStatus' = [accStatus \text{ EXCEPT } ![a] = \text{“stopped”}]$
$\land$ UNCHANGED $\langle status, initVals, pending, execAcks, interface, dState, abortVals, panicAcks, pastPending\rangle$

The full spec

$Init \triangleq$
$\land\ InitProcs$
$\land\ InitAcceptor$
$\land\ interface = InterfaceInit$
$\land\ network = \{\}$
$\land\ abortVals = \{\}$
$\land\ initVals = \{\}$
$\land\ pastPending = \{\}$

$Next \triangleq$
$\lor\ \exists\, p \in P : Inv(p) \lor Ini(p) \lor RcvPanicAck(p) \lor RcvExecAcC(p)$
$\lor\ Panic(p) \lor Abo(p) \lor Res(p)$
$\lor \exists\, a \in Acceptor : WakeUp(a) \lor Exec(a) \lor Stop(a)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Fast \triangleq$ INSTANCE $FastMPGC$

THEOREM $Spec \Rightarrow Fast!Spec$

MODULE *ZLight*

EXTENDS *Generic*, *Library*, *TLCDefs*
INSTANCE *RDR*

CONSTANTS *Initial*, *Leader*, *Follower*
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*

$vars \triangleq \langle status, pending, initVals, execAcks, panicAcks, dState,$
$accStatus, interface, network, abortVals, pastPending\rangle$

$Labels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“pending”}, \text{“panic”}, \text{“aborted”}\}$
$AcceptorLabels \triangleq \{\text{“idle”}, \text{“ready”}, \text{“stopped”}\}$

$Agent \triangleq P \cup Acceptor$
TLC must be able to test members of a set for equality, therefore one cannot have the following set: {1, TRUE}. Since *TLC* can test equality of sequences pointwise starting with the first element, we will make sure that messages are sequences whose first element is a string.
$Msg \triangleq \{\langle \text{“req”}, r\rangle : r \in Req\} \cup \{\langle \text{“execAck”}, s\rangle : s \in S\}$
$\cup \{\langle \text{“panic”}\rangle\} \cup \{\langle \text{“panicAck”}, s\rangle : s \in S\}$
$\cup \{\langle \text{“init”}, s\rangle : s \in S\}$
$\cup \{\langle \text{“leaderInit”}, s\rangle : s \in S\}$
$\cup \{\langle \text{“leaderExec”}, s, p\rangle : s \in S, p \in P\}$
INSTANCE *Network*

$TypeInvariant \triangleq$
$\land \forall p \in P :$
$\land status[p] \in Labels$
$\land pending[p] \in Req$
$\land \forall a \in Acceptor :$
$\land execAcks[p][a] \in \{\{s\} : s \in S\} \cup \{\{\}\}$
$\land panicAcks[p][a] \in \{\{s\} : s \in S\} \cup \{\{\}\}$
$\land \forall a \in Acceptor :$
$\land dState[a] \in S$
$\land accStatus[a] \in AcceptorLabels$

$\land\ initVals \subseteq S$
$\land\ abortVals \subseteq S$
$\land\ pastPending \subseteq Req$

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⟩

 A process can panic at any time because it times out.
Panic(p) ≜
  ∧  status[p] = "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 ∈ Follower ↦ {⟨"leaderInit", s⟩}])
            ∧ dState' = [dState EXCEPT ![rep] = s]
        ELSE ∃ s ∈ S :
            ∧ Rcv(rep, ⟨"leaderInit", s⟩, Leader)
            ∧ dState' = [dState EXCEPT ![rep] = s]
    ∧ UNCHANGED ⟨status, pending, execAcks, interface,
      initVals, abortVals, panicAcks, pastPending⟩

Exec(rep) ≜
  ∧ accStatus[rep] = "ready"
  ∧ IF rep = Leader
      THEN ∃ p ∈ P, req ∈ Req :
       LET newDState ≜ dState[rep] • req IN
         ∧ RcvSnd(rep, ⟨"req", req⟩, p, [x ∈ Follower ∪ {p} ↦
           IF x ∈ Follower
            THEN {⟨"leaderExec", newDState, p⟩}
            ELSE {⟨"execAck", dState[rep] • req⟩}])
         ∧ dState' = [dState EXCEPT ![rep] = newDState]
      ELSE ∃ s ∈ S, p ∈ P :
        ∧    ∃ req ∈ Req : s = dState[rep] • req  don't skip updates
        ∧    RcvSnd(rep, ⟨"leaderExec", s, p⟩, Leader,
          [q ∈ {p} ↦ {⟨"execAck", s⟩}])
        ∧ dState' = [dState EXCEPT ![rep] = s]
  ∧ UNCHANGED ⟨status, pending, execAcks, interface, accStatus,
    initVals, abortVals, panicAcks, pastPending⟩

Stop(a) ≜
  ∧ accStatus[a] = "ready"
  ∧ ∃ p ∈ P : RcvSnd(a, ⟨"panic"⟩, p,
    [q ∈ {p} ↦ {⟨"panicAck", dState[a]⟩}])
  ∧ accStatus' = [accStatus EXCEPT ![a] = "stopped"]
  ∧ UNCHANGED ⟨status, initVals, pending, execAcks, interface, dState,
    abortVals, panicAcks, pastPending⟩
```

The full spec

```
Init ≜
  ∧ InitProcs
  ∧ InitAcceptor
  ∧ interface = InterfaceInit
  ∧ network = {}
```

$\land\ abortVals = \{\}$
$\land\ initVals = \{\}$
$\land\ pastPending = \{\}$

$Next \triangleq$
$\lor\ \exists\, p \in P : Inv(p) \lor Ini(p) \lor RcvPanicAck(p) \lor RcvExecAcC(p)$
$\quad \lor Panic(p) \lor Abo(p) \lor Res(p)$
$\lor\ \exists\, a \in Acceptor : WakeUp(a) \lor Exec(a) \lor Stop(a)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Safe \triangleq$ INSTANCE $SafeMPGC$

THEOREM $Spec \Rightarrow Safe\,!\,Spec$

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 $\triangleq$ INSTANCE *Splitter* WITH
interface $\leftarrow$ *spinterface*,
pc $\leftarrow$ *splitterPc*

splitterVars $\triangleq$ $\langle$*splitterPc*, *x*, *y*, *spinterface*$\rangle$

vars $\triangleq$ $\langle$*v*, *d*, *contention*, *pending*, *pc*, *interface*, *spinterface*, *abortVals*$\rangle$

TypeInvariant $\triangleq$
$\land$ *pc* $\in$ [*P* $\rightarrow$ {"L1", "L2", "L3", "L4", "L5", "L6", "L7", "L8", "L9",
"L10", "COMMITTED", "ABORTED"}]
$\land$ *pending* $\in$ [*P* $\rightarrow$ *Req*]
$\land$ *v* $\in$ *S*
$\land$ *d* $\in$ BOOLEAN
$\land$ *contention* $\in$ BOOLEAN

Init $\triangleq$
$\land$ *pc* = [*p* $\in$ *P* $\mapsto$ "L1"]
$\land$ *d* = FALSE
$\land$ *v* = *Bot*
$\land$ *contention* = FALSE
$\land$ *pending* = [*p* $\in$ *P* $\mapsto$ *Some*(*Req*)]
$\land$ *interface* = *InterfaceInit*
$\land$ *abortVals* = {}

PCFromTo(*p*, *l*1, *l*2) $\triangleq$
$\land$ *pc*[*p*] = *l*1
$\land$ *pc*$'$ = [*pc* EXCEPT ![*p*] = *l*2]

Return(*p*, *o*) $\triangleq$

```
    ∧ 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"
      To be more precise, one should not atomically return or abort and read "contention".
    ∧ 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]
```

$\land$ UNCHANGED $\langle d,\ contention,\ pending,\ interface,\ spinterface,\ abortVals\rangle$

$Step5(p) \triangleq$
$\land\ pc[p] = \text{“L5”}$
$\land$ IF $\neg contention$
THEN
$\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“L6”}]$
ELSE
$\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{“L8”}]$
$\land$ UNCHANGED $\langle v,\ d,\ contention,\ pending,\ interface,\ spinterface,\ abortVals\rangle$

$Step6(p) \triangleq$
$\land\ PCFromTo(p,\ \text{“L6”},\ \text{“L7”})$
$\land\ d' = \text{TRUE}$
$\land$ 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 $\langle v,\ d,\ contention,\ pending,\ spinterface\rangle$

$Step9(p) \triangleq$
$\land\ PCFromTo(p,\ \text{“L9”},\ \text{“L10”})$
$\land\ contention' = \text{TRUE}$
$\land$ UNCHANGED $\langle v,\ d,\ pending,\ interface,\ spinterface,\ abortVals\rangle$

Here we could commit in case v is not Bot, but only with the cstruct version.
$Step10(p) \triangleq$
$\land\ pc[p] = \text{“L10”}$
$\land\ GiveUp(p,\ v)$
$\land$ 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 ≜
  [p ∈ P ↦
    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

$$
\begin{array}{l}
SpInterfaceType \triangleq [\\
\quad resp : [P \to [\\
\qquad output : \text{BOOLEAN} , \\
\qquad flag : \text{BOOLEAN}]], \\
\quad inv : [P \to \text{BOOLEAN}]] \\
SpInterfaceInit \triangleq [\\
\quad resp \mapsto [p \in P \mapsto [\\
\qquad output \mapsto Some(\text{BOOLEAN}), \\
\qquad flag \mapsto Some(\text{BOOLEAN})]], \\
\quad inv \mapsto [p \in P \mapsto Some(\text{BOOLEAN})]]
\end{array}
$$

$$
\begin{array}{l}
InvokeSplitter(p,\ interface,\ newinterface) \triangleq \\
\quad newinterface = [interface \ \text{EXCEPT} \ !.inv = [@ \ \text{EXCEPT} \ ![p] = \neg @]]
\end{array}
$$

$$
\begin{array}{l}
SplitterResponse(p,\ b,\ interface,\ newinterface) \triangleq \\
\quad newinterface = [interface \ \text{EXCEPT} \ !.resp = [@ \ \text{EXCEPT} \ ![p] = [\\
\qquad output \mapsto b, \\
\qquad flag \mapsto \neg @.flag]]]
\end{array}
$$

MODULE *Splitter*

EXTENDS *SplitterInterface*, *Library*

CONSTANT P

VARIABLES
x, y, pc,
interface

$vars \triangleq \langle x, y, pc, interface \rangle$
$Labels \triangleq \{\text{"START"}, \text{"L1"}, \text{"L2"}, \text{"L3"}, \text{"L4"}, \text{"END"}\}$

$TypeInvariant \triangleq$
$\land\ x \in P$
$\land\ y \in \text{BOOLEAN}$
$\land\ pc \in [P \to Labels]$
$\land\ interface \in SpInterfaceType$

$PCFromTo(p, l1, l2) \triangleq$
$\land\ pc[p] = l1$
$\land\ pc' = [pc \text{ EXCEPT } ![p] = l2]$

$Start(p) \triangleq$
$\land\ pc[p] = \text{"START"}$
$\land\ InvokeSplitter(p, interface, interface')$
$\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{"L1"}]$
$\land\ \text{UNCHANGED } \langle x, y \rangle$

$WriteX(p) \triangleq$
$\land\ PCFromTo(p, \text{"L1"}, \text{"L2"})$
$\land\ x' = p$
$\land\ interface' = interface$
$\land\ \text{UNCHANGED } y$
$\land\ \text{UNCHANGED } \langle y, interface \rangle$

$TestY(p) \triangleq$
$\land\ pc[p] = \text{"L2"}$
$\land\ \text{IF } y = \text{TRUE}$
THEN $\land\ SplitterResponse(p, \text{FALSE}, interface, interface')$
$\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{"END"}]$
ELSE $\land\ pc' = [pc \text{ EXCEPT } ![p] = \text{"L3"}]$
$\land\ interface' = interface$
$\land\ \text{UNCHANGED } \langle x, y \rangle$

$WriteY(p) \triangleq$
$\land\ PCFromTo(p, \text{"L3"}, \text{"L4"})$
$\land\ y' = \text{TRUE}$

$\land\ interface' = interface$
$\land$ 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 $\langle x,\ y \rangle$

$Init \triangleq$
$\land\ pc = [p \in P \mapsto \text{“START”}]$
$\land\ x = Some(P)$
$\land\ y = \text{FALSE}$
$\land\ interface = SpInterfaceInit$

$Next \triangleq \exists\, p \in P :$
$Start(p) \lor WriteX(p) \lor TestY(p) \lor WriteY(p) \lor TestX(p)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

B Isabelle/HOL Theories

theory *IOA*
imports *Main*
begin

1 I/O Automata

This theory is inspired by the IOA theory of Olaf Mueller

1.1 Signatures

record *'a signature* =
  inputs::*'a set*
  outputs::*'a set*
  internals::*'a set*

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*|)

1.2 I/O Autoamta

type-synonym
  (*'a*, *'s*) *transition* = *'s* × *'a* × *'s*

record $('a, 's)$ *ioa* =
  asig::$'a$ *signature*
  start::$'s$ *set*
  trans::$('a,'s)$*transition set*

abbreviation *act* $A \equiv$ *actions* (*asig* A)
abbreviation *ext* $A \equiv$ *externals* (*asig* A)
abbreviation *int* **where** *int* $A \equiv$ *internals* (*asig* A)
abbreviation *inp* $A \equiv$ *inputs* (*asig* A)
abbreviation *out* $A \equiv$ *outputs* (*asig* A)
abbreviation *local* $A \equiv$ *locals* (*asig* A)

definition *is-ioa*::$('a,'s)$ *ioa* $\Rightarrow$ *bool* **where**
  is-ioa $A \equiv$ *is-asig* (*asig* A)
    $\wedge$ ($\forall$ *triple* . *triple* $\in$ *trans* $A \longrightarrow$ (*fst o snd*) *triple* $\in$ *act* A)

definition *hide* **where**
  hide A *actns* $\equiv A($|*asig* := *hide-asig* (*asig* A) *actns*|$)$

definition *is-trans*::$'s \Rightarrow 'a \Rightarrow ('a,'s)$*ioa* $\Rightarrow 's \Rightarrow$ *bool* **where**
  is-trans s1 a A s2 $\equiv$ (*s1*,*a*,*s2*) $\in$ *trans A*

notation
  is-trans (- −-−-⟶ - [*81*,*81*,*81*,*81*] *100*)

definition *rename-set* **where**
  rename-set A *ren* $\equiv \{b. \exists\ x \in A$. *ren* b = *Some* $x\}$

definition *rename* **where**
rename A *ren* $\equiv$
  (|*asig* = (|*inputs* = *rename-set* (*inp* A) *ren*,
    outputs = *rename-set* (*out* A) *ren*,
    internals = *rename-set* (*int* A) *ren*|),
   start = *start* A,
   trans = $\{tr. \exists\ x$. *ren* (*fst* (*snd tr*)) = *Some* $x \wedge$ (*fst tr*) $-x-A\longrightarrow$ (*snd* (*snd tr*))$\}$|)

Reachable states and invariants

inductive
  reachable :: $('a, 's)$ *ioa* $\Rightarrow 's \Rightarrow$ *bool*
  for A :: $('a, 's)$ *ioa*
  where
    reachable-0: $s \in$ *start* $A \Longrightarrow$ *reachable* A s
  | *reachable-n*: ⟦ *reachable* A s; $s -a-A\longrightarrow t$ ⟧ $\Longrightarrow$ *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

1.3 Composition of families of ioas

record (*'id*, *'a*) *family* =
ids :: *'id set*
memb :: *'id* ⇒ *'a*

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*, (*'a*, *'s*)*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*, ('*a*, '*s*)*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*, ('*a*, '*s*)*ioa*) *family* ⇒ ('*a*, '*id* ⇒ '*s*)*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*

1.4 Executions and traces

type-synonym
  $('s,'a)pairs = ('s \times 'a)\ list$

type-synonym
— Executions grow to the left
  $('s,'a)execution = ('s,'a)pairs \times 's$
type-synonym
  $'a\ trace = 'a\ list$

record $('a,'s)execution\text{-}module =$
  $execs::('a,'s)execution\ set$
  $asig::'a\ signature$

record $'a\ trace\text{-}module =$
  $traces::'a\ trace\ set$
  $asig::'a\ signature$

fun $is\text{-}exec\text{-}frag\text{-}of::('a,'s)ioa \Rightarrow ('s,'a)execution \Rightarrow bool$ **where**
  $is\text{-}exec\text{-}frag\text{-}of\ A\ ((p\#p'\#ps),\ s) =$
    $(fst\ p' -snd\ p-A\longrightarrow fst\ p \wedge is\text{-}exec\text{-}frag\text{-}of\ A\ ((p'\#ps),\ s))$
| $is\text{-}exec\text{-}frag\text{-}of\ A\ ([p],\ s) = s\ -snd\ p-A\longrightarrow fst\ p$
| $is\text{-}exec\text{-}frag\text{-}of\ A\ ([],\ s) = True$

definition $is\text{-}exec\text{-}of::('a,'s)ioa \Rightarrow ('s,'a)execution \Rightarrow bool$ **where**
  $is\text{-}exec\text{-}of\ A\ e \equiv snd\ e \in start\ A \wedge is\text{-}exec\text{-}frag\text{-}of\ A\ e$

definition *filter-act* **where**
  $filter\text{-}act \equiv map\ snd$

definition *schedule* **where**
  $schedule \equiv filter\text{-}act\ o\ fst$

definition *trace* **where**
  $trace\ sig \equiv filter\ (\lambda\ a\ .\ a \in externals\ sig)\ o\ schedule$

definition *is-schedule-of* **where**
  $is\text{-}schedule\text{-}of\ A\ sch \equiv$
    $(\exists\ e\ .\ is\text{-}exec\text{-}of\ A\ e \wedge sch = filter\text{-}act\ (fst\ e))$

definition *is-trace-of* **where**
  $is\text{-}trace\text{-}of\ A\ tr \equiv$

$(\exists\ sch\ .\ is\text{-}schedule\text{-}of\ A\ sch \wedge tr = filter\ (\lambda\ a.\ a \in ext\ A)\ sch)$

definition *traces* **where**
traces A $\equiv \{tr.\ is\text{-}trace\text{-}of\ A\ tr\}$

lemma *traces-alt*:
shows *traces A* $= \{tr\ .\ \exists\ e\ .\ is\text{-}exec\text{-}of\ A\ e$
$\wedge\ tr = trace\ (ioa.asig\ A)\ e\}$
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* (*fst 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* :: (*'a*,*'s1*)*ioa* ⇒ (*'a*,*'s2*)*ioa* ⇒ *bool* (**infixr** =<| *12*)
where
A =<| *B* ≡ *inp A* = *inp B* ∧ *out A* = *out B* ∧ *traces A* ⊆ *traces B*

1.5 Operations on executions

definition *cons-exec* **where**

cons-exec p e ≡ (*p*#(*fst e*), *snd e*)

definition *append-exec* **where**
append-exec e′ e ≡ ((*fst e′*)@(*fst e*), *snd e*)

fun *last-state* **where**
last-state ([],*s*) = *s*
| *last-state* (*p*#*ps*,*s*) = *fst 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 fst e arbitrary*: *e*)
case *Nil*
from *Nil.prems* **have** *1*:*snd e* ∈ *start A* **by** (*simp add*:*is-exec-of-def*)
from *Nil.hyps* **have** *2*:*last-state e* = *snd 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′* = (*ps*, *snd e*)
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′*)−(*snd p*)−*A*⟶(*fst p*)
by (*simp add*:*is-exec-of-def*) (*cases* (*A*,*p*#*ps*,*snd e*) *rule*:*is-exec-frag-of.cases*, *auto*)
with *ih* **and** *Cons.hyps*(*2*) **show** *?case*
by (*metis fst-conv last-state.simps*(*2*) *prod.exhaust reachable-n*)
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* (*s′*,*a*) *e*)
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 p e*)
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* = *snd 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* −(*snd p*)−*A*⟶ *fst 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* (*p*#(*fst e*), *snd e*) **using** *2*.*prems*(*1*)
by (*metis cons-exec-def pair-collapse trans-from-last-state*)
moreover
have *append-exec e*′ *e* = (*p*#(*fst e*), *snd e*) **using** *2*.*hyps*
by (*metis append-Cons append-Nil append-exec-def*)
ultimately
show *?case* **by** *auto*
next
case (*1 A p p*′ *ps e*′)
have *is-exec-frag-of A* ((*p*#*p*′#*ps*)@(*fst e*),*snd e*)

```
    proof −
      have is-exec-frag-of A ((p'#ps)@(fst e),snd e)
        by (metis 1.hyps 1.prems append-exec-def cons-exec-def
            exec-frag-prefix fst-conv prod-eqI snd-conv)
      moreover
      have fst p' −(snd p)−A⟶ fst 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 = ((p#p'#ps)@(fst e),snd e)
      by (metis 1.hyps(2) append-exec-def)
    ultimately show ?case by simp
  qed
qed

lemma last-state-of-append:
  fixes e e'
  assumes snd 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


theory Simulations
imports IOA
begin
```

2 Definition and soundness of refinement mappings, forward simulations and backward simulations

```
definition refines where
  refines e s a t A f ≡ snd 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) ⇒ ('a,'s1)ioa ⇒ ('a,'s2)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*)) ⇒ (*'a*,*'s1*)*ioa* ⇒ (*'a*,*'s2*)*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* . *snd 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*)) ⇒ (*'a*,*'s1*)*ioa* ⇒ (*'a*,*'s2*)*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* . *snd 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 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 e-A'* **and** *e-B* ≡ *cons-exec* (*t*,*a*) *e-B'*
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 a* # *trace* (*ioa.asig A*) *e-B' 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 a* # *trace* (*ioa.asig A*) *e-A' else trace* (*ioa.asig A*) *e-A'*)
using *2 trace-append-is-append-trace*[*of ioa.asig A e e-A'*]
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

4 Soundness of refinement mappings

lemma *ref-map-execs*:
fixes *A*::(*'a*,*'sA*)*ioa* **and** *B*::(*'a*,*'sB*)*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 fst e-B arbitrary*:*e-B*)
case *Nil*
let *?e-A* = ([], *f* (*snd 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′* = (*ps*, *snd e-B*)
let *?s* = *last-state ?e-B′* **let** *?t* = *fst p* **let** *?a* = *snd p*
have *1*:*is-exec-of B ?e-B′* **and** *2*:*?s*−(*snd p*)−*B*⟶(*fst p*)
using *Cons.prems* **and** *Cons.hyps*(*2*)
by (*simp-all add*:*is-exec-of-def*,

cases (*B*,*p*#*ps*,*snd e-B*) *rule*:*is-exec-frag-of*.*cases*, *auto*,
cases (*B*,*p*#*ps*,*snd e-B*) *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*:*snd 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 e-A′*
have *is-exec-of A ?e-A*
using *ih1 ih3 4 6 append-exec-frags-is-exec-frag*[*of A e-A′ e*]
by (*metis append-exec-def is-exec-of-def snd-conv*)
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*::('*a*,'*sA*)*ioa* **and** *B*::('*a*,'*sB*)*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*

5 Soundness of forward simulations

lemma *forward-sim-execs*:
fixes *A*::('*a*,'*sA*)*ioa* **and** *B*::('*a*,'*sB*)*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 fst e-B arbitrary*:*e-B*)
case *Nil*
have $\bigwedge$ *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* (*snd 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'* = (*ps*, *snd e-B*)
let *?s* = *last-state ?e-B'* **let** *?t* = *fst p* **let** *?a* = *snd p*
have *1*:*is-exec-of B ?e-B'* **and** *2*:*?s*−(*snd p*)−*B*⟶(*fst p*)
using *Cons.prems* **and** *Cons.hyps*(*2*)
by (*simp-all add*:*is-exec-of-def*,
cases (*B*,*p*#*ps*,*snd e-B*) *rule*:*is-exec-frag-of.cases*, *auto*,
cases (*B*,*p*#*ps*,*snd e-B*) *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*:*snd 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 e-A'*
have *is-exec-of A ?e-A*
using *ih1 ih3 4 6 append-exec-frags-is-exec-frag*[*of A e-A' e*]
by (*metis append-exec-def is-exec-of-def snd-conv*)
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*::('*a*,'*sA*)*ioa* **and** *B*::('*a*,'*sB*)*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*

6 Soundness of backward simulations

lemma *backward-sim-execs*:
fixes *A*::('*a*,'*sA*)*ioa* **and** *B*::('*a*,'*sB*)*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 fst 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'* = (*ps*, *snd e-B*)
let *?s* = *last-state ?e-B'* **let** *?t* = *fst p* **let** *?a* = *snd 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−(snd p)−B⟶(fst p)
      using Cons.prems and Cons.hyps(2)
        by (simp-all add:is-exec-of-def,
            cases (B,p#ps,snd e-B) rule:is-exec-frag-of.cases, auto,
            cases (B,p#ps,snd e-B) rule:is-exec-frag-of.cases, auto)
    from 1 have 3:reachable B ?s using last-state-reachable by fast
    obtain e where 4:snd 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' = snd e
        using 1 4 Cons.hyps(1) by (metis fst-conv)
    let ?e-A = append-exec e e-A'
    have is-exec-of A ?e-A
      using ih1 ih3 4 6 append-exec-frags-is-exec-frag[of A e-A' e]
      by (metis append-exec-def is-exec-of-def snd-conv)
    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::('a,'sA)ioa and B::('a,'sB)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
```