5 Speculative Linearizability

5.1 Introduction

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

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

The $SLin(\Delta)[i, i+1]$ automaton models an i^{th} mode instance which behaves *speculatively*, i.e., which only works 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 clients must detect the inconsistency, *abort* their execution of the current mode instance and *switch* to the next mode instance, passing a Δ-state as switch value. When the 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 instance uses the Δ-states received as switch values from the previous instance. The family of I/O automata $SLin(\Delta)$ formally specifies this process and, notably, defines how the execution of a mode should be encoded in the switch values in order for the next mode to continue the execution and ensure that it remains linearizable.

The parameter Δ of the family of I/O automata $SLin(\Delta)$ must be a *recoverable data-type representation*, abbreviated *RDR*, which is a special case of data-type representation. An RDR guarantees that a consistent state can be recovered from a set of different states of the RDR.

The notion of RDR is based on the notion of C-Struct Set proposed in [12] to generalize the Paxos algorithm.

5.2 Related Work

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

The abstraction and compositional properties of Linearizability [10, 13, 14, 8], presented in chapter 3, are useful in simplifying the development of distributed systems: to reason about the safety of a distributed system containing linearizable objects, it suffices to consider only the executions in which the linearizable objects are accessed sequentially, thus abstracting over concurrent accesses of the objects; 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 [4] define communication-closed layers and show that to reason about the safety of algorithms composed of communication-closed layers, one can assume that the layers are sequentially composed, without interleaving. Charron-Bost and Schiper [3] 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 [7], meaning that verifying them on a system containing 5 processes is sufficient to conclude that the system is correct for any number of processes. Emerson and Kahlon derives cutoff bounds [6] for systems whose processes are instances of a generic process template. Examples include a cache coherence protocol. A later paper [5] generalizes the method to networks of heterogeneous processes.

The Abstract framework [9] proposes a reduction theorem that is the main inspiration behind the Speculative Linearizability framework. In the Abstract framework, adaptive algorithms do not optimize the execution of commuting requests and must maintain full execution histories in their data-structures. Inspired by work on the Generalized Consensus problem [12], we have in our turn generalized the Abstract framework to allow optimized execution of commuting requests and to minimize the size of the data-structures that implementations must use. The abstract framework is obtained by instantiating the speculative

linearizability framework for the $Generic$ data type defined in section 3.2.3.

5.3 Recoverable Data-Type Representations (RDRs)

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

We say that a Δ-state d is smaller than a Δ-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 Δ-states is not necessarily a partial order, for example when the transition relation δ has cycles.

A Δ-state d is a *lower bound* of a set of Δ-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 Δ-states ds, when it exists. Also note that the glb of a set of Δ-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 Δ-states, $\preceq$, is antisymmetric.*

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

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

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

TODO: A figure to illustrate the corollary? Hard to depict.

Properties 5.3.1 and 5.3.2 imply that that the set S of Δ-states and the "smaller than" relation form 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 for the successful recovery of an aborted instance of the $SLin$ I/O automaton.

The reader who is familiar with the Generalized Consensus problem [12] will recognize the

similarity between RDRs and C-Struct Sets. Although similar, RDRs have a notion of behavior that includes the outputs that clients receive, whereas C-Struct Sets do not.

We now show that any data type has a RDR and, in particular, we present the *History RDR*, $H^{\#}(D)$, of a data type. Like $Fold(\Delta)$, 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, a Δ-state d is smaller than a Δ-state d' if d is a prefix of d'. Moreover, the greatest lower bound of d and d' is their longest common prefix.

The RDR $Unfold(\Delta)$ is not a very efficient representation because it uses full execution histories. 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

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

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

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

and, for every position j, if $i < j$ and $\langle rs[i], rs[j] \rangle \in \#$, then the permutation σ preserves the order of i and j, $\sigma[i] < \sigma[j]$.

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.

The history data-type representing $H^{\#}(D)$ uses equivalence classes of the dependency relation as states. 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 Δ-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 $\gamma_{\#}$,

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

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.

TODO: Say something rigorous about equivalence classes and stuff in the notation section. Then invoke it to justify the definitions.

We now have the following important property.

Theorem 5.3.1. *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 [12], 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 [15] on trace theory. □

Theorem 5.3.1 is important because, in contrast to $Unfold(\Delta)$, executing commutative re-

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

5.4 Speculative Linearizability

Speculative linearizability is a modular property

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

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 $SLin[1,i]$ where $i > 1$.

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

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

Signature

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

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

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,

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

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

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

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

State Space and Transition Relation

The state of $SLin\,[1,i]$ consists of 4 components, $dState$, tracking the current state of the RDR Δ, $abortVals$, tracking the set of abort values that have been produced so far, 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 $\perp$, $abortVals$ is the empty set, and, for every client p, $status\,[p]$ = "ready" and $pending\,[p]$ is arbitrary. As in the $ModeInst\,(1,p)$ I/O automaton, a client p can be either in status "ready", "pending", or "aborted".

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

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

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

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

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

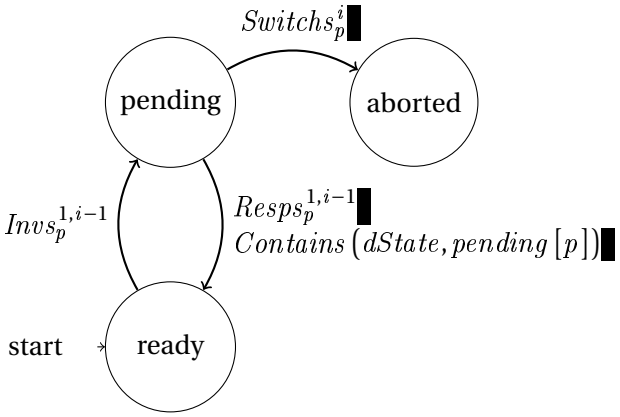

Figure 5.1 – The control flow of a process p in the $SLin\left[1,i\right]$ I/O automaton

an "approximate" but safe estimate of $dState$ from an implementation that has been corrupted by overly optimistic speculative updates.

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

An Important Invariant

Property 5.4.1. *In every reachable state of $SLin\left[1,i\right]$, every abort value $av \in abortValues$ is of the form $dState \star rs$, where $rs \in Seq\left(PendingReqs\right)$.*

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

5.4.2 Linearizability of $SLin$

We see that, ignoring the abort actions, the actions of the $SLin\left[1,i\right]$ 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, $SLin\left[1,i\right]$ implements $NDLin$.

Theorem 5.4.1. *For every $i \in \mathbb{N}$, the projection of $SLin\left[1,i\right]$ 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 $SLin\left[1,i\right]$ to a state t of $NDLin$ such that

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

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

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

Proof sketch. Using corollary 3.4.1, $NDLin \leq Lin$, by transitivity of the implementation relation. □

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

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

Signature

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

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

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

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

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

We see that $SLin\,[i,i+1]$ has the signature of a well-formed mode instance, that the signature of $SLin\,[1,i]$ is compatible with the signature of $SLin\left[i,j\right]$, and that the external signature of $SLin\,[1,i] \times SLin\left[i,j\right]$ is equal to the external signature of $SLin\left[1,j\right]$.

State Space

The state of $SLin\left[i,j\right]$ consists of 6 components, $dState$, tracking the current Δ-state, $intVals$, tracking the set of init values that have been received so far, $abortVals$, tracking the set of abort values that have been produced so far, $initialized$, a boolean, 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.

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

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 $SLin[1,i]$, the initial control flow of a client if not "ready" but "idle".

Transition Relation

Given a state s of $SLin[i,j]$, we define four sets of Δ-states: the set of glbs of init values, G, the set of *safe init values*, $SafeInits$, the set of *choosable values*, $Choosable$, and the set of *safe abort values*, $SafeAborts$. We will see that safe init values are used in the recover action to initialize $dState$, choosable values are used in the $Linearize^i$ action to update $dState$, and safe abort values are used in the $Switch_p^j$ actions as abort values. The intuition behind the definitions presented below are to be found in the proof sketch of the idempotence property of $SLin$.

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

$$G = \{GLB(ivs) : ivs \subseteq initVals\}. \tag{5.10}$$

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

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

We say that a Δ-state d is a *safe Δ-state*, $d \in SafeDStates$, when

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

More formally, the set of safe Δ-states is defined as follows.

$$\begin{aligned} SafeDStates = \{s \in S : dState \preceq s \wedge (\forall av \in abortVals : s \preceq av) \\ \wedge \exists rs \in Seq(PendingReqs) : s = dState \star rs \vee \exists g \in G : s = g \star rs\}. \end{aligned} \tag{5.11}$$

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

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

Formally, if $initialized$ is false, then

$$SafeAborts = \left\{g \star rs : g \in G \wedge rs \in Seq\left(PendingReqs\right)\right\}, \tag{5.12}$$

and if $initialized$ is true, then

$$\begin{aligned} SafeAborts = \big\{&s \in S : dState \preceq s \\ &\wedge \exists rs \in Seq\left(PendingReqs\right) : s = dState \star rs \vee \exists g \in G : s = g \star rs\big\} \end{aligned} \tag{5.13}$$

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

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

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

5.4.4 Idempotence of $SLin$

We have shown in section 5.4.2 that $SLin$ is linearizable. To prove that $SLin$ is a modular property, we still need to show that $SLin$ is idempotent and well-formed. We now address

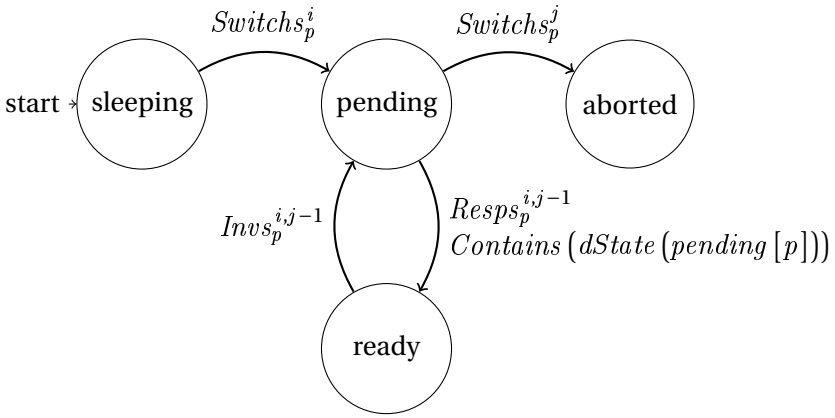

Figure 5.2 – The control flow of a process p in the $SLin\left[i,j\right]$ I/O automaton when $i > 1$.

idempotence. The invariants and refinement proof sketch below should help the reader understand the definitions of the previous section.

Theorem 5.4.2 (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 this result we first need to establish a few invariants of the I/O automaton $SLin\left[1,i\right] \times SLin\left[i,i+1\right]$.

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

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

$$PendingReqs' = PendingReqs\left(s_2\right) \cup \left\{pending\left[p\right] : status\left(s_1\right)\left[p\right] = \text{"pending"}\right\} \tag{5.14}$$

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

Lemma 5.4.1 (Invariant 1)**.** *If* $initialized\left(s_2\right)$ *is false, then* $PendingReqs'$ *is a equal to* $PendingReqs\left(s_1\right)$*.*

Lemma 5.4.2 (Invariant 2)**.** *If* $initialized\left(s_2\right)$ *is false, then for every safe init value* $iv \in SafeInits\left(s_2\right)$*, there exists a sequence of pending requests* $rs \in PendingReqs'$ *such that* $iv = dState\left(s_1\right) \star rs$*.*

Lemma 5.4.3 (Invariant 3)**.** *If* $initialized\left(s_2\right)$ *is false, then for every safe abort value* $av \in SafeAborts\left(s_2\right)$*, there exists a sequence of pending requests* $rs \in PendingReqs'$ *such that* $av = dState\left(s_1\right) \star rs$*.*

Lemma 5.4.4 (Invariant 4)**.** *If* $initialized\left(s_2\right)$ *is true and the* Δ*-state* d *is such that*

- $dState\left(s_2\right) \leq d$ *and*
- *there exists* $g \in G\left(s_2\right)$ *and a sequence of pending requests* $rs \in Seq\left(PendingReqs\left(s_2\right)\right)$ *such that* $d = g \star rs$*,*

then there exists a sequence of requests $rs' \in Seq\left(PendingReqs'\right)$ *such that* $d = dState\left(s_2\right) \star rs'$*.*

The invariants 2, 3, and 4 follow from the conjunction of the invariant of $SLin\,[1,i]$ presented in the previous section (property 5.4.1), the consistency property of recoverable data-type representations, and the first invariant.

Let us now sketch the proof of theorem 5.4.2

Theorem 5.4.2 (Idempotence of $SLin$)**.** *The family of I/O automata* $\{SLin\,[i,j] : i,j \in \mathbb{N}\}$ *is idempotent.*

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

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

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

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

The most interesting cases are those of the $Recover^i$ action, the $Switch^j$ abort action of $SLin\,[i,i+1]$, and the $Linearize^i$ action of $SLin\,[i,i+1]$.

□

5.4.5 $SLin$ is a modular property

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

Theorem 5.4.3 ($SLin$ 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.*

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

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. □

Finally, we can prove our main theorem.

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

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

5.4.6 Proving Idempotence Mechanically

In an effort to make the results of this thesis trustworthy, we have mechanically proved in Isabelle/HOL the idempotence of a restricted version of the speculative linearizability property. We present our proof in this chapter.

Isabelle/HOL [16] 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 [18]. Isabelle/HOL allows writing and interactively proving statements in higher order logic. All proofs are checked by a small, highly trusted kernel of inference rules. A large library of derived proof rules and theorems is available and several packages provide automated setup for higher level concepts such as records, recursive and co-recursive data-types [19], recursive functions, modular organisation of specifications with locales [11], etc. The Isar proof language [20] 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 [17], and Sledgehammer [1], which can call external automatic provers and SMT solvers [1] and reconstruct the obtained proofs in Isabelle/HOL. Moreover, the Nitpick tool [2] can search for counterexamples to putative theorems.

We have proved the idempotence theorem for a specification I/O automaton ALM which is close to the $SLin$ I/O automaton except that the data type is fixed to the $Generic$ data type presented in section 3.2.3 and that its behavior is restricted in a few corner cases.

Consider the representation Δ of the $Generic$ data-type presented in section 3.2.3. Remember that in an execution of Δ, the state of Δ is the sequence of requests, without the duplicates, that have been executed up to this point. Moreover, the output contained in a response is the

current state. The data-type representation Δ is a recoverable data-type representation: the "less than" relation on states is the prefix relation on sequences, and the glb of a set of Δ-states is their longest common prefix.

The I/O automata $ALM[i,j]$, for $1 \leq i < j$, is very similar to the I/O automaton $SLin(\Delta)[i,j]$ both in structure and in behavior. The $ALM[1,i]$ I/O automaton, for $1 < i$, has the same set of traces as $SLin[1,i]$. The set of traces of the $ALM[i,j]$ I/O automaton, for $1 < i < j$, is a strict subset of the set of traces of $SLin[i,j]$ because the abort actions of $ALM[i,j]$ are more restricted. In $ALM[i,j]$, when the boolean $initialized$ is true, the safe abort values are of the form $d = dState \star rs$, where rs is a sequence of pending requests. However, in $SLin[i,j]$, the abort values can also be of the form $d = g \star rs$, where $g \in \{GLB(is) : is \subseteq initVals\}$, rs is a sequence of pending requests, and $dState \leq d$. If there is an init value which is strictly bigger than $dState$ and which cannot be obtained by appending pending requests to $dState$, then some safe abort values of $SLin$ are not safe abort values of ALM.

The difference between the ALM I/O automata and the $SLin$ I/O automata is not significant and they both have the same structure and rely on the same invariants. However we have found out by model checking our specifications that, in a corner case, the $Quorum$ algorithm violates the more restricted abort actions of the ALM I/O automata.

The Isabelle/HOL proof shows that $ALM[1,i] \times ALM[i,j]$ implements $ALM[1,j]$, for $1 < i < j$. The refinement mapping is essentially the same as in the proof of theorem 5.4.2. We prove the refinement mapping correct with the help of 15 state invariants about the composite automaton. The proof is written in the structured proof language Isar and consists of roughly 500 proof steps (lines containing the keywork "by"). With the specification, it forms a total of 1600 lines of Isabelle/HOL code.

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

We conclude the chapter by a few remarks on our experience with Isabelle/HOL. It is extremely time consuming for a relatively novice user to formalize and prove in Isabelle/HOL a theory that is not well-understood beforehand. The problem is that Nitpick and the other debugging tools available in Isabelle are not able to check high level properties like the idempotence or linearizability of $SLin$. Only deeply nested proof steps can be debugged in Isabelle/HOL. As a result, many errors where discovered late in the development and ultimately, although ALM was proved idempotent after a lot of effort, it was found inadequate for proving $Quorum$. After this experience, we formalized all of our results in TLA+ and we were able to check all our claims, end to end, with TLC before even attempting to write a proof. Many errors were eliminated in the process, which culminated in a few month to the theory presented in this thesis. In contrast, our first development took more than a year and resulted in a mechanically checked proof of a property which is not exactly the right one in

practice. In conclusion, even though experienced users may be able to use Isabelle/HOL effectively, the learning curve is still too steep for an outsider. However, debugging tools that allow quick prototyping are extremely useful and if integrated with Isabelle/HOL could allow a much broader audience to use it.

5.5 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 distributed implementation of the state of a RDR in a way that leads to increased performance under some optimistic assumptions and to the corruption of the state otherwise. If the state of the system is corrupted by an overly optimistic update, then the 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 model of computation. To do so, we will present speculatively linearizable adaptive algorithms that efficiently implement any data type. We will also see in chapter 7 that speculative linearizability can be applied to the shared-memory model.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] Lawrence C. PAULSON. "Isabelle: The Next 700 Theorem Provers". In: *CoRR* cs.LO/9301106 (1993).

[19] 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.

[20] 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.

Croix Rouges 12
1007 Lausanne, Switzerland

+41 78 669 64 32
giuliano.losa@epfl.ch

Giuliano Losa

Education

08/2009-02/2014 (expected)
EPFL, Switzerland, PhD student in Computer Science, 5th year.

Supervised by Rachid Guerraoui and Viktor Kuncak.

Thesis title: Modularity in the Design of Robust Distributed Algorithms.

2007-2009 EPFL, Switzerland, Master in Computer Science.

2005-2009 Supélec, France, Master in Electrical Engineering.

2003 Baccalauréat Scientifique, option mathématiques, mention très bien.

Work Experience

09/2008-03/2009
IBM T.J. Watson Research Center, USA, Data-Intensive Systems and Analytics Group. Master thesis.

Design and specification of the SPL programming language.

Design and implementation of a distributed object store, using C++.

07/2007-08/2007
C.E.A., France, Study and port of a hard real-time operating system on Linux.

Publications

Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. "Speculative linearizability". In: *PLDI*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 55–66. DOI: 10.1145/2254064.2254072.

Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. "Abortable Linearizable Modules". In: *The Archive of Formal Proofs*. Ed. by Gerwin Klein, Tobias Nipkow, and Lawrence Paulson. Formal proof development. http://afp.sf.net/entries/Abortable_Linearizable_Modules.shtml, 2012.

Dan Alistarh et al. “On the cost of composing shared-memory algorithms”. In: *SPAA*. Ed. by Guy E. Blelloch and Maurice Herlihy. ACM, 2012, pp. 298–307. DOI: 10.1145/2312005.2312057

Giuliano Losa et al. “CAPSULE: language and system support for efficient state sharing in distributed stream processing systems”. In: *DEBS*. Ed. by François Bry et al. ACM, 2012, pp. 268–277. DOI: 10.1145/2335484.2335514

Martin Hirzel et al. *SPL Stream Processing Language Specification, IBM Research report RC24897*. Tech. rep. IBM, 2009

Languages

French: native speaker.

English: excellent.

Italian: fluent.

German: basic.

Extra-curricular activities

2006-2007 President of “Supélec Rézo”, the student association in charge of the computer network of Supélec’s campus in Gif-Sur-Yvette.