2 Specifying Distributed Systems

2.1 Introduction

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

This chapter is an introduction to the basic concepts of the theory of I/O automataand of the TLA+ language. In the rest of the thesis, we use the theory of I/O automata [13] for informal discussions and the TLA+ [11] language for formal specifications. In chapter 8, which we describe the formalization and mechanical proof of one of our results in the Isabelle/HOL [21] interactive theorem prover.

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

In an effort to provide a trustworthy theory of adaptive distributed systems, we have formalized our work in the TLA+ language and we have checked the correctness of our results with the TLC model checker [24]. In section 2.4.6, we describe how to translate I/O automata specifications in TLA+ in order to use the TLC model checker.

There are many other specification frameworks targeting the description of distributed systems and their properties. Some frameworks are well-known as frameworks while others are better known by the name of their main component. Let us cite the BIP framework (Behavior, Interaction, and Priority) [2], the I/O-automata framework [10], TLA+ [11] (the Temporal Logic of Actions), Reactive Modules [1], Promela and the SPIN model checker [9], the NuSMV model checker [4], Bigraphical Reactive Systems [16], Abstract State Machines [3], and process calculi like CSP [8], the π-calculus [17, 18], and Petri nets [22].

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 and $Last(e)$ for e_n. We use $\circ$ for sequence concatenation, $\langle e_1, \ldots, e_n \rangle \circ \langle f_1, \ldots, f_m \rangle = \langle e_1, \ldots, e_n, f_1, \ldots, f_m \rangle$. Appending an element e to a sequence es is 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 A.4.

I/O automata were first introduced by Lynch and Tuttle [13] to model asynchronous distributed systems. The theory of I/O automata is also described in details in chapter 8 of Lynch's book [12] , 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 [14].

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 [20, 19], parts of which are still maintained in the Archive of Formal Proofs. Second, there is the IOA Toolkit [10], which is composed of a formal specification of the IOA language, a simulator [23], a verifier based on the LP theorem prover [6], and a tool for generating Java programs from IOA specifications [7]. 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 8.2.

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 [12] 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*, $Inputs\,(sig)$, the set of input actions of Sig, $Outputs\,(sig)$, the set of output actions, and $Internals\,(sig)$, the set of internal actions. The set of actions of a signature, noted $Acts\,(sig)$, is the union of all three components, whereas the set of external actions, noted $Ext\,(sig)$, 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 $\langle s, a, s' \rangle$ 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 $\langle s, a, s' \rangle \in \delta(\Sigma)$. When $\langle s, a, s' \rangle$ 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 $Inputs(Sig)$, $Outputs(A)$ for $Outputs(Sig)$, $Internals(A)$ for $Internals(Sig)$, $Ext(A)$ for $Ext(Sig)$, $Acts(A)$ for $Acts(A.sig)$, $Start(A)$ for $Start(\Sigma)$, $\delta(A)$ for $\delta(\Sigma)$, and $States(A)$ for $States(\Sigma)$.

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

Execution and schedules

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

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

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

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

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

We say that an action a is enabled in a state s if there exists a transition, $\langle s, a, s' \rangle$, 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 $s \overset{t}{\Longrightarrow}_A s'$ 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

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

  where if a is an action of A_i, then $\left\langle s_i, a, s'_i \right\rangle$ 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 $Ass\left[i,j\right]$ 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 $A_1 \le B_1$ and $A_2 \le B_2$ then $A_1 \times A_2 \le B_1 \times B_2$.

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* $i \in 1..n$, $A_i \le B_i$*, then*

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

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

2.3.3 Hiding and Projection

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

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

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

Theorem 2.3.2. *If* $A \le B$*, then* $hide(A, S) \le hide(B, S)$

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

$$proj(A, S) = hide(A, Acts(A) \setminus S) \tag{2.9}$$

Theorem 2.3.3. *If* $A \le B$*, then* $proj(A, S) \le proj(B, S)$

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 using a refinement mapping in conjunction with history variables or by using a forward simulation. There are other types of simulation proofs, using prophecy variables or backward simulations. However we only use history history variables an forward simulations in this thesis. For a thorough explanation of simulation proofs methods, we refer the reader to Lynch and Vaandrager [14].

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 nonempty sets H and $H_0 \subseteq H$ such that

$$A_H = \langle Sig, \langle S \times H, S_0 \times H_0, C, \delta_H\rangle\rangle \tag{2.10}$$

where δ_H is such that

1. if $\left\langle \langle s,h\rangle, a, \left\langle s', h'\right\rangle\right\rangle$ is a transition of δ_H, then $\left\langle s, a, s'\right\rangle$ is a transition of δ;
2. if $\left\langle s, a, s'\right\rangle$ is a transition of δ, then, for every $h \in H$, there exists $h' \in H$ such that $\left\langle \langle s,h\rangle, a, \left\langle s', h'\right\rangle\right\rangle$ is a transition of 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 $f[s] \overset{\langle a\rangle}{\Longrightarrow}_B f\left[s'\right]$;
  - if $a \notin Ext(B)$, then $f[s] \overset{\langle\rangle}{\Longrightarrow}_B f\left[s'\right]$.

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.*

A forward simulation from A to B is a *relation* r such that:

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

Theorem 2.3.6. *Consider two I/O automata A and B which have the same external signature. If r is a forward simulation from A to B, then A implements B.*

Forward simulations have the same power as the combination of a history variable and refinement mapping: one can prove that A implements B using a forward simulation if and only if one can prove it using a refinement mapping in conjunction with a history variable. A proof of this result appears in [14]. However, in practice, a proof may be easier with one or the other method. We will use theorem 2.3.6 and corollary 2.3.1 throughout the thesis to prove implementation relations between I/O automata. Backward simulations, not presented here, are formalized in the Isabelle/HOL theory called "Simulations" which can be found in appendix A.4.

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 [11] or the article of Merz [15], and we would be unable to better explain TLA+. Therefore, instead of explaining TLA+ in details, we will only highlight its main features and give a few examples that we hope will suffice for the reader to understand our discussion. Note that the TLA+ examples are typeset with the TLA+ typesetter and do not follow the notation introduced earlier.

We have used TLC within the TLA Toolbox, which offers a user-friendly Integrated Development Environment for TLA+ specifications. The TLA Toolbox provides a graphical interface to edit, check, and prove specifications correct and the TLC model checker is integrated in the toolbox and allows fast and visual debugging of specifications. All the parameters of TLC can be control with the GUI and the graphical trace explorer simplifies the analysis of error traces. All our TLA+ specification can be found in appendix A. TLA+ specifications can be also be proved correct and mechanically checked in the TLA Toolbox with TLAPS [5]. However TLAPS is still in development at the time of writing and we have preferred using Isabelle/HOL for writing mechanically-checked proofs.

2.4.1 A Basic Example

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

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

$Next1 \triangleq x' = x + 1$

$Init1 \triangleq x = 0$

$Spec1 \triangleq Init1 \land \Box Next1$

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

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

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

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

2.4.2 The Implementation Relation

Consider the following specification $Spec2$.

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

$Next2 \triangleq \land\, y' = \neg y$

$\qquad\qquad \land \text{ IF } y \text{ THEN } x' = x + 1 \text{ ELSE } x' = x$

$Spec2 \triangleq Init2 \land \Box Next2$

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

To define implementation in terms of trace inclusion we need to allow the specification $Spec1$ to "stutter", i.e., take steps where x does not change while the other variables are updated arbitrarily. Therefore, in TLA+, specifications must be of the form $Init \land \Box [Next]_{vars}$, 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 $Next \lor \left(v_1' = v_1 \land \cdots \land v_n' = v_n\right)$.

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

$Init1 \triangleq x = 0$

$Next1 \triangleq x' = x + 1$

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

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

$Next2 \triangleq \land\, y' = \neg y$

$\qquad\qquad \land\, \text{IF } y \text{ THEN } x' = x + 1 \text{ ELSE } x' = x$

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

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

2.4.3 Refinement Mappings

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

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

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 $Inv2 \land Next2 \Rightarrow Inv2'$

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 $Inv2 \land Next2 \Rightarrow Next1$

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

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

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

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

Suppose that $vars1$ and $vars2$ are disjoint. In this case the behaviors satisfying $F1 \land F2$ are composed of four kinds of steps: steps satisfying $Next1 \land Next2$, called *joint steps*, steps satisfying $Next1 \land vars2' = vars2$, steps satisfying $Next2 \land vars1' = vars1$, and steps satisfying $vars1' = vars1 \land vars2' = vars2$. If $vars1$ and $vars2$ intersect, then every step modifying a variable of $vars1 \cap vars2$ must be a joint step. The specification of two communicating systems can therefore be obtained by conjoining two specifications that change common variables representing the interface between the two specifications. Note that, in the resulting specification, the two communicating components may take joint steps even when they do not communicate (when both only update variables not in $vars1 \cap vars2$). In contrast, two I/O automata in a composite I/O automaton take joint steps only when communicating.

This concludes our brief presentation of TLA+. We have not addressed many important topics, like using history and prophecy variables in refinement proofs, proving temporal properties, etc.. We refer the reader to the works of Lamport [11] and Merz [15].

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 the transformation is sound, i.e., that $[\![A]\!] \Rightarrow [\![B]\!]$, in TLA+, implies that the I/O automaton A implements the I/O automaton B. We assume that A and B have the same external signature; otherwise we already know that $A \leq B$ does not hold.

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

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

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

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

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

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

We use the operator

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

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

Finally, we define

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

and, similarly, we define

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

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

$$(\boldsymbol{\exists}\, s_A, int_A : \llbracket A \rrbracket) \Rightarrow (\boldsymbol{\exists}\, s_B, int_B : \llbracket B \rrbracket) \tag{2.16}$$

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 $\llbracket A \times B \rrbracket$ in terms of $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$, for example as $\llbracket A \rrbracket \wedge \llbracket B \rrbracket$. This does not work because I/O automata take joint steps only when emitting an action that is common to both I/O automatonand otherwise evolve independently, whereas in $\llbracket A \rrbracket \wedge \llbracket B \rrbracket$ joint steps can occur even when no common action is emitted.

We can prevent unwanted joint actions by conjoining to the specification formulas stating that joint steps must represent a common action. Therefore in $\llbracket A \rrbracket$ we separate the ext variable in two variables $common$ and ext_A and in $\llbracket B \rrbracket$ we separate the ext variable in two variables $common$ and ext_B.

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

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

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

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

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

Finally, define

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

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

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

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

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 presented the theory of I/O automata and the TLA+ language.

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

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

Bibliography

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

[14] 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.

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

[16] 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.

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

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.