Chapter 9
Concurrent Systems

Was mir an deinem System am besten gefällt? Es ist so unverständlich wie die Welt. (What do I like most about your system? It is as incomprehensible as the world.) — Franz Grillparzer

The previous chapters have modeled computer programs mainly as "black boxes" that accept some input and produce some output; the internal operation of a program to achieve this behavior was considered as "irrelevant" to the external observer and thus deliberately hidden. While this view is indeed adequate to model sequential program executions, it fails if we wish to consider *concurrent systems*, i.e., systems where multiple components (cores, processors, computers) execute multiple activities (threads, processes) in parallel. The various components may interact with each other by synchronization respectively communication; such systems of components that react to influences from other components are also called *reactive systems*.

In this chapter, we will consider the formal modeling and reasoning about such systems. This presentation is based on system models expressed in the language of first-order logic as well as on property specifications expressed in linear temporal logic, an extension of first-order logic that is able to talk not only about a single state or a fixed number of states but about arbitrary sequences of such states.

The first part of this chapter focuses on the formal modeling of concurrent systems. Section 9.1 lays the groundwork by introducing the core concept of *labeled transition systems* that represent the formal semantics of concurrent systems; these models can be described by formulas in first-order logic. Sections 9.2 and 9.3 continue by presenting two languages of *shared systems* (systems that consist of closely coupled threads interacting via shared variables in a single store) respectively *distributed systems* (systems composed of loosely coupled processes that interact via well-defined interfaces by exchanging messages); we formally define a denotational semantics of both languages as labeled transition systems.

Further on, we turn our attention to the formal specification and verification of properties of labeled transition systems. Section 9.4 introduces the language of *linear temporal logic* that is adequate to express many relevant system properties; we also provide a semantic characterization of these properties as *safety* properties, *liveness* properties, or properties that are mixtures of both aspects. In Section 9.5, we consider in more detail the verification of *invariance*, a particular class of safety properties that is also fundamental to reasoning about other properties; the notion of invariance can be considered as a generalization of the partial correctness of sequential programs.

Likewise, we discuss in Section 9.6 the verification of *response*, a particular class of liveness properties that can be considered as a generalization of the notion of the termination of sequential programs. Finally Section 9.7 concludes by discussing the formal *refinement* of systems as the basis of a systematic methodology for concurrent system development.

9.1 Labeled Transition Systems

Systems, States, Actions, and Runs

For modeling and analyzing concurrent systems, it is not only the two states at the start and at the end of the execution that matter; we rather have to consider also all intermediate states. In other words, we have to deal with complete *system runs* of the following kind:

$$s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$$

Such a run embeds the sequence $[s_0, s_1, s_2, \ldots]$ of those states that arise in an execution of the system; as in Chapter 7, each state is represented by a mapping of variables to values. Every "step" $s \xrightarrow{l} s'$ of the run represents the execution of an *action* with label l that performs a *transition* from one intermediate state s, the *prestate* of the transition, to another intermediate state s', the transition's *poststate*. In contrast to sequential computer programs, systems may intentionally not terminate, thus the state sequence may not only be finite but also infinite.

A system run exposes all those states that are "visible" among the components of the system; however, a transition $s \xrightarrow{l} s'$ may also involve some "hidden" intermediate states occurring between prestate s and poststate s', provided that these intermediate states do not affect the interactions among the components. For example, consider the execution of the following system *XY1* (whose notation will be formally introduced in Section 9.2):

```
shared XY1 {
  var x:nat, y:nat;
  init { x := 0; y := 0 }
  action inc { x := x+1; y := y+x }
}
```

The states of this system are pairs of natural numbers $x \in \mathbb{N}$ and $y \in \mathbb{N}$. Starting with the initial state $[x \mapsto 0, y \mapsto 0]$, the values are repeatedly incremented by an action *inc*, which gives rise to the following system run:

$$[x \mapsto 0, y \mapsto 0] \xrightarrow{inc} [x \mapsto 1, y \mapsto 1] \xrightarrow{inc} [x \mapsto 2, y \mapsto 3] \xrightarrow{inc} \ldots$$

In each step of this run both x and y are simultaneously incremented, "hiding" the action's intermediate state where x has been already incremented but y has not yet

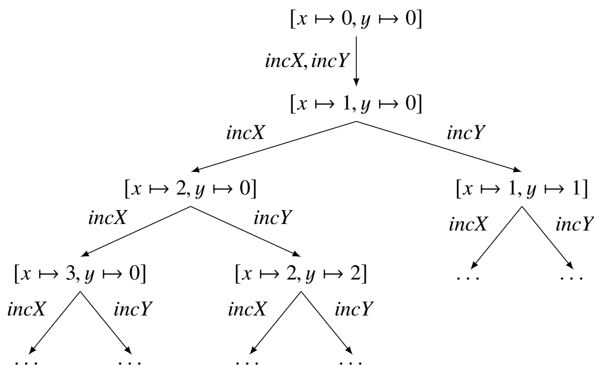
© 2020 Wolfgang Schreiner.

changed. Each action thus represents an "atomic" unit of execution whose effects are *simultaneously* exposed to the external observer. The "granularity" of a system run may thus be adapted by combining multiple commands into a single action respectively by separating commands into multiple actions.

Above example system contains a single action, which naturally models a system with a single component. The execution of such a system, starting in some given initial state, clearly yields a predetermined sequence of successor states; we call such a system *deterministic*. However, if a system consists of multiple components that may execute concurrently with each other, we can model such a system only by multiple actions, at least one per component. Then in a given system state it can be generally not predetermined which of the actions is executed next, which gives rise to multiple possible system runs with different sequences of states; we call such a system *nondeterministic*. As an example, consider the following system *XY2*:

```
shared XY2 {
  var x:nat, y:nat;
  init { x := 0; y := 0 }
  action incX { x := x+1 }
  action incY { y := y+x }
}
```

This system has two actions *incX* and *incY* incrementing x and y, respectively. In every system state it is possible that each of these actions is executed; thus there are always two possible successor states. All possible executions of the system can be represented by a labeled directed graph as follows:



Every node in this graph represents a *state* of the system; every edge represents a *transition* from a prestate (denoted by the source of the edge) to a poststate (denoted by the target of the edge); the edge label denotes the *action* that performs the transition. We can draw (a finite portion of) the graph by drawing, starting with a node for every initial system state (depicted by nodes with incoming arrows without sources and labels), an edge for every transition, which (potentially) leads to new states for which new graph nodes are created from which again new edges arise. The graph is not necessarily a tree, because in general the same poststate may be reached from different prestates; the graph may even contain cycles that describe how from a certain state a number of transitions lead to the same state again.

© 2020 Wolfgang Schreiner.

Every path in the graph starting from an initial state (i.e., a node representing such a state) describes a potential *run* of the system. Every depicted graph node represents a *reachable* state, i.e., a state that may arise in some system run. If a run leads to a node with no outgoing edge (we will see later that this is possible), the run is finite. A run is infinite, if it passes through infinitely many different states or if it infinitely often returns to the same state again.

Labeled Transition Systems

We are now going to formalize the concepts we have introduced above, in particular a mathematical model of "systems".

Definition 9.1 (Labeled Transition Systems). Given a set L of elements which we call *labels* and a set S of elements which we call *states*, we define the set

$$LTS^{L,S} \subseteq \mathsf{space}{:}\mathsf{Set}(S) \times \mathsf{init}{:}\mathsf{Set}(S) \times \mathsf{next}{:}\mathsf{Set}(L \times S \times S)$$

of *labeled transition systems* over L and S as follows:

$$LTS^{L,S} := \{\langle \mathsf{space}{:}S_0, \mathsf{init}{:}I, \mathsf{next}{:}R\rangle \mid S_0 \subseteq S \wedge I \subseteq S_0 \wedge R \subseteq L \times S_0 \times S_0\}$$

Thus a labeled transition system (*LTS*) *lts* over L and S consists of a *state space* $S_0 := lts.\mathsf{space} \subseteq S$, an *initial state condition* $I := lts.\mathsf{init} \subseteq S_0$, and a *(labeled) transition relation* $R := lts.\mathsf{next} \subseteq L \times S_0 \times S_0$.

This definition differentiates between the set S of system states and the state space S_0 of a concrete LTS; this difference is only due to technical reasons in the subsequent formal definitions and may be mainly ignored.

Given a fixed LTS with label set L, state space S_0, and transition relation R, we write $s \xrightarrow{l} s'$ to indicate the transition $R\langle l, s, s'\rangle$ where $l \in L$ is the *label* of the *action* performing the transition, $s \in S_0$ is its *prestate*, and $s' \in S_0$ is its *poststate*.

A LTS gives rise to system runs of the following kind.

Definition 9.2 (System Runs). Given a set S, we define the set

$$Run^S := S^* \cup S^\omega$$

as the set of *system runs* (short: *runs*) over S. Thus a run $r \in Run^S$ is a finite or infinite sequence of values from S.

© 2020 Wolfgang Schreiner.

System Analysis in TLA$^+$ and RISCAL

In this chapter we present two instances of the logical model of the client-server system introduced in Examples 9.4 and 9.18 of the previous chapter; we will verify these instances with the help of model checking software.

The first instance is developed in the *TLA$^+$ toolbox* [76] which is based on Lamport's *Temporal Logic of Actions* [74] and is described in the book [75] and the chapter [88]. This toolbox is an integrated development environment for writing system specifications in the TLA$^+$ modeling language, including the TLC model checker and the TLAPS proof system. The language and system have been used for modeling and analyzing industrial hardware and software systems, e.g., Amazon Web Services (AWS). TLA$^+$ system models are essentially (unlabeled) transition systems as presented in Chapter 9 expressed as linear temporal logic (LTL) formulas constructed from an initial state condition and a transition relation. The TLC model checker can fully automatically verify the correctness of finite instances of such systems with respect to arbitrary correctness properties expressible as LTL formulas. For infinite state systems, the interactive TLPS prover can be used to construct invariant-based proofs of safety properties as described in Chapter 9, using internal tactics or external SMT solvers or provers (such as Isabelle) as backends. In our presentation, however, we will focus on the TLC checker.

The second instance is developed in the *RISCAL* software [132, 133] that was introduced on page 479. RISCAL also supports the modeling of shared and distributed systems as described in Sections 9.2 and 9.3. We demonstrate the verification of a safety property of such systems, by checking the property in all reachable states and by checking invariant-based verification conditions.

The specifications used in this chapter can be downloaded from the URLs

```
https://www.risc.jku.at/people/schreine/TP/software/systems/System.tla
https://www.risc.jku.at/people/schreine/TP/software/systems/System.txt
```

and loaded by executing from the command line the following commands:

```
toolbox System.tla &
RISCAL System.txt &
```

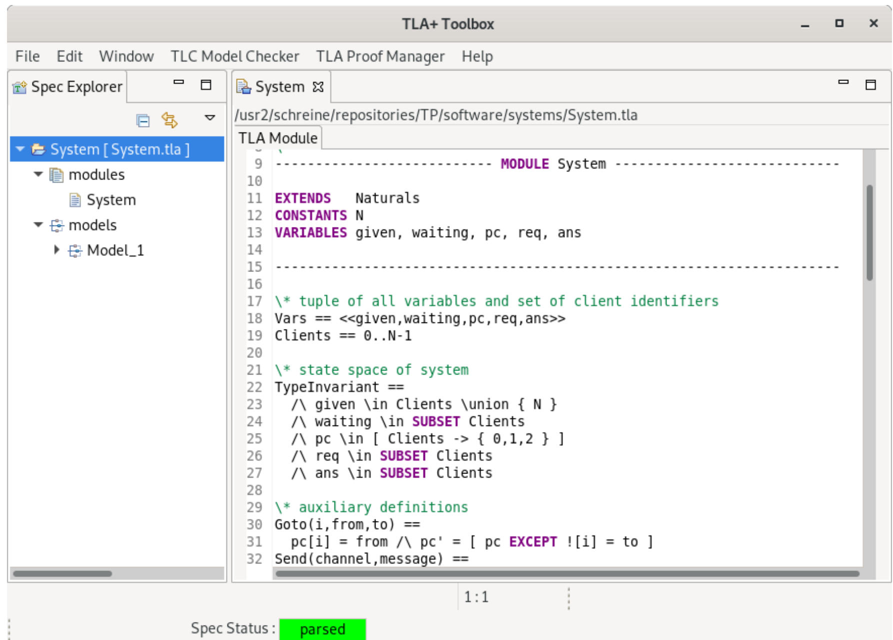


Fig. 9.11 The TLA+ Toolbox

The TLA+ ToolBox

When starting the toolbox from the command line with the name of the specification file `System.tla` as an argument, a graphical user interface is displayed as shown in Figure 9.11. This file essentially describes the model of the client-server system described in Example 9.4 as a module `System`:

```
--------------------------- MODULE System ---------------------------
EXTENDS   Naturals
CONSTANTS N
VARIABLES given, waiting, pc, req, ans
```

The module starts with the import of another module `Naturals` that contains some auxiliary operations on the domain of natural numbers; then it introduces a model constant N that denotes the number of clients and it declares those variables whose values represent the state of the system. Now we capture by the following definition the list of system variables under the name $Vars$ for later reference:

```
Vars == <<given,waiting,pc,req,ans>>
```

Every such definition introduces an "operator" that describes a named entity (the left side of the definition) by an expression (a term or a formula on the

© 2020 Wolfgang Schreiner.

right side of the definition). It is easiest to think of an operator as a syntactic abbreviation where any later occurrence of the operator name is replaced by its defining expression. Operators may also have parameters and then be instantiated by arbitrary argument expressions; they can be considered as the functions respectively predicates of TLA$^+$.

TLA$^+$ has no static type system, but we may define a predicate that describes the expected domain of these variables (the invariance of this property will be later verified with the TLC checker):

```
Clients == 0..N-1
TypeInvariant ==
  /\ given \in Clients \union { N }
  /\ waiting \in SUBSET Clients
  /\ pc \in [ Clients -> { 0,1,2 } ]
  /\ req \in SUBSET Clients
  /\ ans \in SUBSET Clients
```

As shown in this definition, conjunctions respectively disjunctions may be expressed not only by the logical infix connectives /\ and \/ but also as "itemized lists" where the connectives serve as labels at the beginnings of the list items; this simplifies the reading of specifications.

The initial state condition of the system is defined as a predicate I:

```
I ==
  /\ given = N
  /\ waiting = {}
  /\ pc = [ i \in Clients |-> 0 ]
  /\ req = {}
  /\ ans = {}
```

For a concise definition of the transition relation we introduce a couple of auxiliary predicates:

```
Goto(i,from,to) ==
  pc[i] = from /\ pc' = [ pc EXCEPT ![i] = to ]
Send(channel,message) ==
  message \notin channel /\ channel' = channel\union{message}
Receive(channel,message) ==
  message \in channel /\ channel' = channel\{message}
```

Any plain variable reference such as *pc* refers here to the value of the system variable in the prestate of a transition while the primed reference *pc'* refers to the value of the variable in the poststate. With the help of these predicates, we may define the transition relation *RC(i)* of client i as follows:

```
RC0(i) ==
  \/ Goto(i,0,1) /\ Send(req,i)    /\ ans' = ans
  \/ Goto(i,1,2) /\ Receive(ans,i) /\ req' = req
  \/ Goto(i,2,0) /\ Send(req,i)    /\ ans' = ans
RC(i) == RC0(i) /\ given' = given /\ waiting' = waiting
```

Similarly we describe the transition relation *RC(i)* of the server when processing a message from client i:

© 2020 Wolfgang Schreiner.

```
ClientServer ==
  /\ I /\ [][R]_Vars
  /\ \A i \in Clients: WF_Vars(RC(i)) /\ WF_Vars(RS(i))
  /\ \A g \in Clients:
       SF_Vars(\E i \in Clients: Receive(req,i) /\ i = given /\
         GiveWaiting(g) /\ pc' = pc)
```

Here we require *strong* fairness for every message from a client g in the set *waiting* of the server: if it is infinitely often possible that, by the receipt of a message i of some client that was given the resource (and that thus by the message gives back the resource) it is possible to give the resource to the waiting client g, this must be eventually also done. It is really necessary to demand strong (not just weak) fairness, because, if the resource is given to another client, the enabling condition is disabled until the message is returned again. The predicate described in the fairness condition is exactly the enabling condition of the second transition of the server:

```
RS0(i) ==
  \/ ...
  \/ (i = given /\ \E g \in Clients: GiveWaiting(g))
  \/ ...
RS(i) == Receive(req,i) /\ RS0(i) /\ pc' = pc
```

Indeed with this refined definition of the server also for N=3 the check of property *NotStarve* succeeds.

RISCAL

This section shows how to verify in RISCAL invariance conditions for shared and distributed systems as presented in Sections 9.2 and 9.3.

Shared Systems

First we are going to model the RISCAL version of the shared system presented in Example 9.4. For this we introduce the number N of clients aund various auxiliary types:

```
val N:ℕ; axiom minN ⇔ N ≥ 1;
type Client = ℕ[N-1]; type Client0 = ℕ[N]; type PC = ℕ[2];
```

The core declaration of the system describes its state space and its initial state as follows:

```
shared system ClientServer1
{
  var given: Client0; var waiting: Set[Client];
  var pc: Array[N,PC]; var req: Set[Client]; var ans: Set[Client];
```

© 2020 Wolfgang Schreiner.

```
    init()
    {
      given := N; waiting := ∅[Client];
      pc := Array[N,PC](0); req := ∅[Client]; ans := ∅[Client];
    }
    ...
  }
```

This declaration is extended by the transitions of every client *i*:

```
action cask(i:Client) with pc[i] = 0 ∧ i ∉ req;
{ pc[i] := 1; req := req ∪ {i}; }
action cget(i:Client) with pc[i] = 1 ∧ i ∈ ans;
{ pc[i] := 2; ans := ans \ {i}; }
action cret(i:Client) with pc[i] = 2 ∧ i ∉ req;
{ pc[i] := 0; req := req ∪ {i}; }
```

Likewise we add the transitions of the server:

```
action sget(i:Client) with i ∈ req ∧ given = N ∧ i ∉ ans;
{ req := req \ {i}; given := i; ans := ans ∪ {i}; }
action swait(i:Client) with i ∈ req ∧ given ≠ N ∧ given ≠ i;
{ req := req \ {i}; waiting := waiting ∪ {i}; }
action sret1(i:Client) with i ∈ req ∧ given = i ∧ waiting = ∅[Client];
{ req := req \ {i}; given := N; }
action sret2(i:Client,j:Client)
with i ∈ req ∧ given = i ∧ j ∈ waiting ∧ j ∉ ans;
{ req := req \ {i}; given := j;
  waiting = waiting \ {j}; ans := ans ∪ {j}; }
```

Furthermore, we annotate the system with the invariant that shall be satisfied by every reachable state of the system, i.e., the mutual exclusion property:

```
invariant ¬∃i1:Client,i2:Client with i1 < i2. pc[i1] = 2 ∧ pc[i2] = 2;
```

Now checking the system with N=4 quickly demonstrates its correctness:

```
Executing system ClientServer1.
340 system states found with search depth 213.
Execution completed (46 ms).
```

However, if we invalidate the correctness of the system by commenting out the command `given := i` in server action *sget* we get the following result:

```
Executing system ClientServer1.
ERROR in execution of system ClientServer1: evaluation of
  invariant ¬(∃i1:Client, i2:Client with i1 < i2. ...);
at line 28 in file System.txt:
  invariant is violated
The system run leading to this error:
  0:[given:4,waiting:{},pc:[0,0,0,0],req:{},ans:{}]->cask(0)->
  1:[given:4,waiting:{},pc:[1,0,0,0],req:{0},ans:{}]->cask(1)->
  2:[given:4,waiting:{},pc:[1,1,0,0],req:{0,1},ans:{}]->sget(0)->
  3:[given:4,waiting:{},pc:[1,1,0,0],req:{1},ans:{0}]->cget(0)->
  4:[given:4,waiting:{},pc:[2,1,0,0],req:{1},ans:{}]->sget(1)->
```

© 2020 Wolfgang Schreiner.

```
  5:[given:4,waiting:{},pc:[2,1,0,0],req:{},ans:{1}]->cget(1)->
  6:[given:4,waiting:{},pc:[2,2,0,0],req:{},ans:{}]
ERROR encountered in execution.
```

This message produces a counterexample run, i.e., a run that leads to a state violating the invariant. The system may be further annotated with more invariants (see Example 9.18):

```
invariant ∀i:Client with i = given.
  (pc[i] = 0 ∧ i ∈ req) ∨ (pc[i] = 1 ∧ i ∈ ans) ∨
  (pc[i] = 2 ∧ i ∉ req ∧ i ∉ ans);
invariant ∀i:Client with i ∈ waiting.
  i ≠ given ∧ pc[i] = 1 ∧ i ∉ req ∧ i ∉ ans;
invariant ∀i: Client with i ∈ req. i ∉ ans;
invariant ∀i: Client with i ∈ ans. given = i;
invariant ∀i: Client with pc[i] = 0. i ∉ ans ∧ (i ∈ req ⇒ i = given);
invariant ∀i: Client with pc[i] = 1. i ∈ req ∨ i ∈ waiting ∨ i ∈ ans;
invariant ∀i: Client with pc[i] = 2. i = given;
```

Also these invariants can be quickly checked:

```
Executing system ClientServer1.
340 system states found with search depth 213.
Execution completed (39 ms).
```

Actually, these invariants are inductive and imply the mutual exclusion property; as shown below, they can be used to prove the correctness of the system.

Distributed Systems

Now we demonstrate the RISCAL version of the distributed system presented in Example 9.10. For this we introduce (in addition to the entities defined at the beginning of this section) the size B of the message buffers associated the actions of the server:

```
val B:ℕ; axiom minB ⇔ B ≥ 1;
```

The actual system declaration then looks as follows:

```
distributed system ClientServer2
{
  component Server
  {
    var client: Client0;
    init() { client := N; }
    action[B] request(i:Client) with client = N;
    { client := i; send Client[i].enter(); }
    action[B] giveback(i:Client) { client ≔ N; }
  }
  component Client[N]
  {
    var req: ℕ[1]; var use: ℕ[1];
```

© 2020 Wolfgang Schreiner.