Teaching Transition Systems and Formal Specifications with TLA$^+$

Philippe Mauran Philippe Quéinnec Xavier Thirioux
Université de Toulouse
Institut de Recherche en Informatique de Toulouse
2 rue Camichel, BP 7122
F-31071 Toulouse Cedex 7, France
{mauran,queinnec,thirioux}@enseeiht.fr

Abstract

We present here our experience with teaching two courses using TLA$^+$. The first course concerns state transition systems, and the second one is about formal specifications. In the first course, TLA$^+$ is used to describe, reason about, and analyze transition systems. The second course deals with refinement, simulation and bisimulation, and TLA$^+$ is used to check refinements.

1 Context

The courses are taught in two different "engineer" degrees (equivalent to graduate or master degrees) at INPT/ENSEEIHT[1], one in computer science and applied mathematics (80 students), the other one in computer science and networking (15 students). In both cases, students have a solid scientific background with rather good mathematics bases. However, they have no previous exposition to formal specification and modeling. These courses were first taught in 2005, and have been taught each year since then.

All four members of the teaching unit belong to IRIT/ACADIE team[2] which studies methods and tools to verify and certify distributed embedded critical systems. As such, we have a good knowledge both of distributed systems, and of formal techniques. Moreover, two of us are regular users of TLA$^+$ for our research activities.

2 Why TLA$^+$?

We had done previous attempts with other formalisms (Unity, Promela, B...) but none were truly satisfactory. The main reason is that tools were either not available or too complex to master, or that the formalism was too specialized.

Our research experience with TLA$^+$ led us to believe that it could be a good formalism for teaching: light and rather simple, but rigorous; complete (w.r.t to temporal logic and transition systems); modular; and with automatic tools (TLC).

Neither courses are pure TLA$^+$ courses, TLA$^+$ is mainly a tool to express and manipulate formal concepts. Hence, only the relevant parts of TLA$^+$ for a given course are taught.

[1] http://www.enseeiht.fr
[2] http://www.irit.fr/-Equipe-ACADIE-

3 Courses Contents

3.1 Transition Systems

This course[3] is concerned with the specification, modeling, and validation of systems, especially concurrent systems. State transition systems are used as the basis for modeling. Temporal logics (CTL, LTL) are used to specify safety, liveness, and fairness properties. Verification methods (model checking, axiomatic proofs) are used to validate models.

This course is taught in 15 lessons of 1h45: 10 lectures with exercises, and 5 assisted labs. During its presentation, this course mixes formal subjects (transition systems, temporal logics) and "applied" subjects, where TLA$^+$ is used to express the formal concepts previously seen. The presentation of TLA$^+$ is along the lines of Lamport's book [1], using different examples: we first present transition systems and actions, then temporal logic and properties, composition (modules), and refinement.

As students are also trained to checking and assisted proof environments (notably Coq [2] and Why [3]) in a previous course, our approach is to put forward TLA$^+$'s possibilities as a modeling tool rather than the verification environment provided by the TLA$^+$ Toolbox. Thus, throughout exercises, we focus on describing and specifying transition systems, on giving students an intuition of temporal operators, and on expressing systems properties into temporal logic. The use of TLC during lab sessions allows to keep this focus on specification, and to bind clearly, in a logical framework, the specification of systems in terms of transitions and fairness, and their specification in terms of safety and liveness properties. We present and illustrate the semantics of temporal operators, as well as the main proof rules, but we do not require their practical use by students, for the reasons given above. Lastly, we propose a rather standard approach to develop specifications, based on refining and incremental transformations that gradually introduce more details into the system model. On this occasion, we give the students a first exposure to building systems by refining and composing modules, but these topics are more completely covered in the formal specifications course that follows.

3.2 Formal Specifications

This course introduces the formal notion of refinement in open systems, by presenting labeled transition systems with (un)observable events. Simulation and bisimulations relations are defined. Modules and their execution semantics are presented, and refinement between modules is studied. The course has 14 lessons of 1h45, with three nearly equal parts: simulation/bisimulation of transition systems, CCS, refinement.

A in-house small framework based on TLA$^+$ is used in labs to verify refinement properties of module-like code. This framework uses game semantics to describe client's and module's behaviors: the client chooses which procedure it wants to invoke and the value of its input parameter (records are used if we want to emulate a multi-parameter procedure); then the module realizes the procedure by changing its internal state and setting the output value. When choosing a procedure and its input value, the client must ensure that the precondition of the procedure is valid, and inversely, the module must ensure the procedure postcondition. Each of these steps is implemented as a TLA$^+$ action (see annex A for a full example), and an execution is an alternation between an arbitrary client action and the corresponding module action.

[3]Slides, in French, are available here: `http://queinnec.perso.enseeiht.fr/Ens/ens-st.html`.

A refinement is then another client/module TLA$^+$ implementation which must ensure:

- The client's actions are less constraining: the abstract client's actions simulate the concrete client's actions;
- Conversely, the concrete module's actions simulate the abstract module's actions.

At last, a module is considered implementable when it is without deadlock (w.r.t the initial specification) and its actions are deterministic.

3.3 Token Ring Mutual Exclusion

As an illustration, we present the main lines of an implementation of mutual exclusion in a distributed system, by means of a token moving around a ring.

3.3.1 Abstract Version

We start with an abstract version, specifying mutual exclusion between N processes (or sites).

- Each site has an identifier, which is a natural number between 0 and $N - 1$.
- The system state is represented by one or more arrays, indexed by the sites identifiers. The element i of each array stands for a local variable of site i.
- The transitions are defined as TLA$^+$ actions, parameterized by the identifiers of the sites.

This approach is kept throughout each version.

As regards the specification development, we start with defining state variables, then expected properties, then actions, and lastly the *Next* and *Spec* predicates. At this point, we may have to tune a little bit properties or actions, in particular when we state fairness properties. Concerning our example, given the current level of detail, we have to set a strong fairness constraint on the action that allows to enter the critical section. This is an opportunity to illustrate the notion of strong fairness, and to prepare the next step. Properties are limited to mutual exclusion (safety), absence of deadlock (weak liveness), and absence of starvation (strong liveness).

3.3.2 Centralized Version

This version introduces the token, as a global variable, the value of which represents the identifier of the site that holds the token. The liveness and safety properties related to the token's motion are stated, along with the corresponding actions which, again, are parameterized by the sites' identifiers. We note that a token moving along a ring is stronger than necessary, a fair moving token (which infinitely often visits any site) is enough. Usually, at first, the students do not avoid the starvation caused by omitting the forced transfer of the token when a process leaves its critical section, but they find this bug rather easily if we put the focus on the action that releases mutual exclusion. When we consider fairness constraints, we can lay the stress on the fact that we can drop the strong fairness on the actions that allow to enter the critical section, as the global mutual exclusion condition can be detected by a local one, namely the presence of the token.

3.3.3 Distributed Token

This step simply introduces a representation of the token that is closer to a distributed version: the token is represented as an array, indexed by the sites' identifiers. Our purpose is to emphasize the relationship between a centralized representation and a distributed one. Refinement equivalence (via bisimulation) is verified.

3.3.4 Introducing Communication Channels

This step introduces an abstract representation for the communication medium, as a set of channels. Each channel is a link between a site and its successor in the ring. An array of sequences of booleans, indexed by the channels' identifiers is added, in order to represent the sending of a message holding the token. The transfer of the token between each site and its in-going and out-going channels is specified as extra actions, and new properties are added to describe the refinement.

3.3.5 Labs

Lab sessions lay the stress on building a relevant representation of the system to be modeled, and on stating liveness and fairness properties, which are unfamiliar to the students. During labs, the students use TLC to check that *Spec* implies the system's properties (or not). The readability of traces is appreciated, although locating the actions corresponding to state changes remains rather laborious.

4 Empirical Feedback

4.1 Students' Stumbling Blocks

- Fairness is very difficult. This may be the most important point which is not understood by a significant part of the students. Students tend to fully ignore it or to abuse it (adding strong fairness everywhere). We have found that it helps to first introduce trace semantics and temporal logics (LTL, CTL) without any reference to TLA$^+$. Then various kinds of fairness (simple fairness on states, conditional fairness on states, weak and strong fairness on transitions) are explained as a filtering of the traces. At last, fairness in TLA$^+$ is explained with this simple concepts.
- Non-determinism is difficult: even $x' \in S$ is like magic. Non-determinism is seen in previous courses (e.g. an introduction to automata theory) but TLA$^+$ courses are actually the first ones where non-determinism is heavily and practically used. During a large part of the initial lessons, we have to repeat that TLA$^+$ actions are *predicates*, and we purposely use non-determinism to stress it.
- Choose is believed to be like random(). Once again, the rigorous mathematical definition helps to explain the difference.
- Quantifiers are believed easier than set theory: $\forall x \in S : \exists y \in S' : p(x) = y$ is more used than the mathematically equivalent $p(S) \subseteq S'$
- Axiomatic proofs are manageable with regard to invariants. However, liveness formal proofs are too complex.

- The difference between checked properties (i.e. invariants, leadsto) and module specification (i.e. actions) is not clear: students want the checked properties to behave like constraints on the module (especially the usual *TypeInvariant*).
- Students are used to rely on a type checker to catch minor errors (e.g. using $\in$ instead of $\subseteq$). They found the absence of a (even crude) type checker to be cumbersome.

4.2 Achievements

- By first teaching transition systems and transition predicates, students easily grasp that an action is not an assignment. There is enough distance between TLA$^+$ and standard programming languages (e.g. Java or C) so that confusions are mitigated. Moreover it is easy to switch between TLA$^+$ and basic transition systems. The theory of transition systems offer a pure and simple formal description, while TLA$^+$ offers an expressive language to describe these systems.
- LTL is easy and rather intuitive, CTL is not. This is not a surprise to anyone used to both logics!
- Basic TLA$^+$ is simple enough to be quickly understood.
- Complex notions (e.g. fairness, non-determinism) are accessible.

4.3 Tools

As the courses were introduced in 2005, they were based on the command-line tools: TLC, tlatex. TLC has proved to be an *essential* tool for student to experiment. TLC is exclusively used in model-checking mode, never in simulation mode. When dealing with the invalidation of a safety property, its trace is generally sufficient to understand the cause. The fact that it is the shortest trace which invalidates the property helps a lot. When dealing with the invalidation of a liveness property, TLC does not necessarily show the shortest trace, and students are sometimes unsettled by this.

Surprisingly, a difficulty occurs when the model is buggy and yields a too small state space. This happens with basic typos (e.g. $x = x + 1$ instead of $x' = x + 1$) or hasty cut-and-paste (e.g. giving $x' = 1 \land \ldots \land x' = 2$). In this case, safety properties are satisfied, and students are either omitting liveness properties or finding it hard to understand why TLC is unhappy. Then debugging unfortunately relies on `PrintT` statements to discover which actions are always false.

As any automatic tool, the drawback of using TLC is its systematic use to check any combination of properties and actions until no error occurs. Students tend to consider that an absence of error from TLC is a proof of the correctness of their module with regard to the informal specification of the problem, whereas it is only a proof of correctness with regard to their supplied properties. Therefore, some students use a "winning" strategy which consists in deleting or restricting invalid properties rather than correcting the module. Fortunately, the execution time of TLC means that students quickly discover that thinking before running TLC pays in the long term.

In 2012, we plan to experiment with the TLA$^+$ Toolbox. Our own experience lets us believe that the closer integration between the traces and the text of the model will help students to understand an execution.

PlusCal was considered out of scope: our goal is to teach transition systems and formal specification from a mathematical and logical point of view. By its objectives themselves,

Pluscal's algorithmic nature hides too much the underlying mathematical abstract machine.

5 Conclusion

At the end of the two courses, we observe that students have learned to formally specify and verify small systems, with various degrees of abstraction, varying from temporal logic properties (e.g. LTL with $\Box, \Diamond$ and no prime variable) to non-deterministic or implementable TLA$^+$ module specification. Moreover, they have acquired a better understanding of specification transformations (especially state and code refinement) and of relations between specifications (simulation, bisimulation). Thus, we believe that the students have learned useful concepts for their future career.

However, this success is mitigated by two points. First, given the short duration of the courses and the tools limitations with regard to large models, only small models and algorithms are studied, e.g. classical concurrent and distributed algorithms. Moreover, model checking is only attempted with small state space models (a few thousands states). Secondly, formal methods are not yet widely used in the industry. In that sense, the two courses have more sense from an educational point of view, than from an utilitarian one.

References

[1] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[2] The Coq proof assistant. `http://coq.inria.fr/`.

[3] Why3 – where programs meet provers. `http://why3.lri.fr/`.

A Formal Specification with Game Semantics: An Example

This module has two operations: one to acquire a ticket, one to release it. The client can ask for a ticket as long as all have not been given, and it can release any acquired ticket. The module must not give the same ticket twice. More efficient refinements (e.g. using a counter) are then developed and checked.

Pre_X actions are client's actions, and *Act_* actions are module's actions. In the module's actions, *used* is its internal state before the operation, *used_p* is the state after the operation. A run alternates between *ClientContract* and *ModuleContract*.

MODULE *Tag*

EXTENDS *Naturals*

CONSTANTS N

$CHOICES \triangleq \{\text{“acquire”}, \text{“Release”}\}$
$\text{STATE} \triangleq \text{SUBSET } (1 \,..\, N)$

$Init(used) \triangleq used = \{\}$

$Pre\_Acquire(param, used) \triangleq$
$\quad \wedge used \neq 1 \,..\, N$
$\quad \wedge param = \text{“\_\_NO\_DATA”}$
$Act\_Acquire(param, used, used\_p, result) \triangleq$
$\quad \wedge result \in (1 \,..\, N \setminus used)$
$\quad \wedge used\_p = used \cup \{result\}$

$Pre\_Release(param, used) \triangleq$
$\quad \wedge param \in used$
$Act\_Release(param, used, used\_p, result) \triangleq$
$\quad \wedge result = \text{“\_\_NO\_DATA”}$
$\quad \wedge used\_p = used \setminus \{param\}$

$ClientContract(choice, param, used) \triangleq$
$\quad \vee (choice = \text{“Acquire”} \wedge Pre\_Acquire(param, used))$
$\quad \vee (choice = \text{“Release”} \wedge Pre\_Release(param, used))$

$ModuleContract(choice, param, used, used\_p, result) \triangleq$
$\quad \wedge (choice = \text{“Acquire”} \Rightarrow Act\_Acquire(param, used, used\_p, result))$
$\quad \wedge (choice = \text{“Release”} \Rightarrow Act\_Release(param, used, used\_p, result))$