# TLA+ Case Study: A Resource Allocator

Stephan Merz

**To cite this version:**

Stephan Merz. TLA+ Case Study: A Resource Allocator. [Intern report] A04-R-101 || merz04a, 2004, 20 p. inria-00107809

**HAL Id: inria-00107809**
**https://inria.hal.science/inria-00107809**

Submitted on 19 Oct 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# TLA$^+$ Case Study: A Resource Allocator

Stephan Merz
INRIA Lorraine & LORIA, Nancy, France
Stephan.Merz@loria.fr

August 17, 2004

**Abstract**

This note presents a case study for the specification and analysis of reactive systems in TLA$^+$. It illustrates available verification techniques and describes some pitfalls to avoid when writing formal models. It is mainly intended as a tutorial to the TLA$^+$ language and tools and was initially developed during a Summer School in Slovakia in June 2004.[1]

## 1 Problem description

The purpose of the system to be specified is to manage a (finite) set of resources that are shared among a number of client processes. Informally, the requirements for the system can be stated as follows:

1. A client that currently does not hold any resources and that has no pending requests may issue a request for a set of resources.

   We require that no client should be allowed to "extend" a pending request, possibly after the allocator has granted some resources. A single client process might concurrently issue two separate requests for resources by appearing under different identities, and therefore the set of "clients" should really be understood as identifiers for requests, but we will not make this distinction here.

2. The allocator may grant access to a set of available (i.e., not currently allocated) resources to a client.

   Resources can be allocated in batches, so an allocation need not satisfy the entire request of the client; hopefully, the client can already work with a subset of the resources it requested.

3. A client may release some resources that it holds.

   Similarly to allocation, clients may return just a subset of the resources they currently hold, freeing them for allocation to a different process.

4. Clients are required to eventually free the resources they hold once their entire request has been satisfied.

The system should be designed such that it ensures the properties of

[1]This case study was inspired by a model written by Michel Charpentier and students at the University of New Hampshire.

**safety:** no resource is ever allocated to two different clients and

**liveness:** every request issued by some client is eventually satisfied.

This note discusses possible specifications of this system in TLA$^+$ [3] at a relatively high level of abstraction, without premature commitment to implementation decisions. It also describes the use of tools to analyze and verify the models. Along the way, it discusses some potential pitfalls of using formal specification techniques such as TLA$^+$. Basic familiarity with the TLA$^+$ notation is assumed.

## 2 A first model

We start with a high-level specification of the resource allocator. Abstract specifications are easier to understand and validate than detailed ones; they are also more easily analyzed using tools such as model checkers or theorem provers. For our case study, we abstract from many details. For example, we not specify how the allocator and the clients communicate, or how the data is represented. Such aspects can be introduced by later refinement steps after the overall protocol has been modeled, and we will give an indication of how to do this in section 5. What exactly can and should be left out in a high-level model is of course specific to the problem; it also requires judgment that only experience can teach.

A first specification of the resource allocator appears in module *SimpleAllocator* in Fig. 1. It is based on the standard TLA$^+$ module *FiniteSets* and declares the constant parameters *Clients* and *Resources* that represent the sets of client processes and resources; the latter is assumed to be finite. (We do not require the set *Clients* to be finite because the model is correct without this assumption: only a finite number of clients will have issued a request at any point.)

The module declares two variables: as expressed by the typing invariant[2], *unsat* is a function that associates to each client the set of resources that it has requested but not yet received. Similarly, the variable *alloc* is a function associating to each client the set of resources that have been allocated to it. The constant *available* is defined to equal the set of resources that are not currently allocated to any client.

The following definitions introduce the initial predicate and the elementary actions of the model. The state predicate *Init* asserts that no resources have been requested or allocated. The action $Request(c,S)$ represents client $c$ issuing a request for the set $S$ of resources. As stated in requirement (1), this action is enabled only if client $c$ currently has no pending requests and if it does not hold any resources. We also require $S$ to be non-empty; obviously, issuing a request for the empty set of resources is uninteresting.[3] The effect of the action $Request(c,S)$ is to record the request in the array *unsat*; the variable *alloc* is left unchanged.

The action $Allocate(c,S)$ represents allocation of the set $S$ of resources to client $c$. Again, we require $S$ to be non-empty. More importantly, the action states that the resources in $S$ should not only be available but also that $c$ should have requested them, a requirement that does not appear in the informal description (2). Informal requirements

[2]TLA$^+$ is an untyped formalism [4], and the module formally simply defines *TypeInvariant* to be some state predicate. It is, however, good engineering practice to assert the intended types of state variables, and we will verify type correctness when analysing the model.

[3]Formally, such a step would simply be a stuttering action, invisible to the TLA$^+$ specification, and dropping the requirement $S \neq \{\}$ results in an equivalent system specification. We have added it essentially to simplify the interpretation of counter-examples produced by the TLC model checker.

MODULE *SimpleAllocator*

EXTENDS *FiniteSet*
CONSTANTS *Clients, Resources*
ASSUME *IsFiniteSet(Resources)*
VARIABLES
  *unsat*, $unsat[c]$ denotes the outstanding requests of client $c$
  *alloc* $alloc[c]$ denotes the resources allocated to client $c$

---

$TypeInvariant \triangleq$
$\quad \land\ unsat \in [Clients \rightarrow \text{SUBSET}\ Resources]$
$\quad \land\ alloc \in [Clients \rightarrow \text{SUBSET}\ Resources]$

$available \triangleq$ set of resources free for allocation
$\quad Resources \setminus (\text{UNION}\ \{alloc[c] : c \in Clients\})$

---

$Init \triangleq$ initially, no resources have been requested or allocated
$\quad \land\ unsat = [c \in Clients \mapsto \{\}]$
$\quad \land\ alloc = [c \in Clients \mapsto \{\}]$

$Request(c, S) \triangleq$ Client $c$ requests set $S$ of resources
$\quad \land\ unsat[c] = \{\} \land alloc[c] = \{\} \land S \neq \{\}$
$\quad \land\ unsat' = [unsat\ \text{EXCEPT}\ ![c] = S]$
$\quad \land\ \text{UNCHANGED}\ alloc$

$Allocate(c, S) \triangleq$ Set $S$ of available resources are allocated to client $c$
$\quad \land\ S \neq \{\} \land S \subseteq available \cap unsat[c]$
$\quad \land\ alloc' = [alloc\ \text{EXCEPT}\ ![c] = @ \cup S]$
$\quad \land\ unsat' = [unsat\ \text{EXCEPT}\ ![c] = @ \setminus S]$

$Return(c, S) \triangleq$ Client $c$ returns a set of resources that it holds.
$\quad \land\ S \neq \{\} \land S \subseteq alloc[c]$
$\quad \land\ alloc' = [alloc\ \text{EXCEPT}\ ![c] = @ \setminus S]$
$\quad \land\ \text{UNCHANGED}\ unsat$

$Next \triangleq$ The system's next−state relation
$\quad \exists c \in Clients, S \in \text{SUBSET}\ Resources :$
$\qquad Request(c, S) \lor Allocate(c, S) \lor Return(c, S)$

$vars \triangleq \langle unsat, alloc \rangle$

---

$SimpleAllocator \triangleq$ The complete high−level specification
$\quad \land\ Init \land \Box[Next]_{vars}$
$\quad \land\ \forall c \in Clients : \mathrm{WF}_{vars}(Return(c, alloc[c]))$
$\quad \land\ \forall c \in Clients : \mathrm{SF}_{vars}(\exists S \in \text{SUBSET}\ Resources : Allocate(c, S))$

---

Figure 1: First model of the resource allocator.

are bound to be incomplete, and the design process has to identify and resolve open issues. In some cases, it may be desirable to give clients access to resources they have not requested. Should we wish to allow such behavior, we should simply write $S \subseteq \mathit{available}$; it would not make much difference for the issues to be discussed in this note.

Finally, the action $\mathit{Return}(c, S)$ models client $c$ returning the resources in $S$; as stated in the informal requirement (3), we do not require that $c$ returns all resources that it is holding at once.

The system's next-state relation *Next* is simply defined as the disjunction of the actions introduced above, for any client $c$ and any set $S$ of resources. The system specification is represented by the formula *SimpleAllocator*; it is of the standard form

$$\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land L$$

where $L$ is a conjunction of fairness conditions, which we examine next. The first fairness condition asserts that no client should keep its resources forever. There are several ways to express this; we could for example have written

$$\forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\exists S \in \textsc{subset}\ \mathit{Resources} : \mathit{Return}(c, S))$$

without affecting the correctness of the protocol. Formally, $\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$ implies that the two conditions are equivalent—but the formal proof of this equivalence is not trivial.

The second fairness condition is perhaps more interesting. Because resources can be allocated to different clients and thus become temporarily unavailable, weak fairness is not sufficient. One of several equivalent ways of defining strong fairness of action $\langle A \rangle_v$ in TLA is

$$\mathrm{SF}_v(A) \;\equiv\; \Box(\Box\Diamond\textsc{Enabled}\ \langle A \rangle_v \Rightarrow \Diamond\langle A \rangle_v),$$

and thus our fairness condition asserts that the allocator must eventually allocate some resources to client $c$ if this is repeatedly possible. It is a good exercise to try and convince yourself that this is a reasonable fairness requirement for the allocator process.

## 3 Analysis of the model

Whereas programs can be compiled and executed, TLA$^+$ models can be validated and verified. In this way, one gains confidence that a model faithfully reflects the intended system, and that it can serve as a basis for more detailed designs, and ultimately for implementations. Although review and inspection remain the preferred means for validating a model, tools can assist in that process. In particular, simulation can explore some traces allowed by the model, possibly detecting deadlocks or violations of invariants. Deductive tools such as model checkers and theorem provers assist in the formal verification of properties. TLC, the TLA$^+$ model checker, is a powerful and very usable tool for verification and validation, and we will illustrate its use for our example in section 3.1. More ambitious is verification using theorem-proving techniques, and we briefly discuss a hybrid approach in section 3.2.

### 3.1 Analysis by model checking

Figure 2 contains four correctness properties, beyond the type correctness predicate *TypeInvariant*, that should be satisfied by the allocator. The predicate *ResourceMutex*

$ResourceMutex \triangleq$ the same resource is never allocated to different clients
$\quad \forall c_1, c_2 \in Clients : c_1 \neq c_2 \Rightarrow alloc[c_1] \cap alloc[c_2] = \{\}$
$ClientsWillReturn \triangleq$ clients will return resources when requests have been fulfilled
$\quad \forall c \in Clients : unsat[c] = \{\} \leadsto alloc[c] = \{\}$
$ClientsWillObtain \triangleq$ every requested resource will eventually be allocated
$\quad \forall c \in Clients, r \in Resources : r \in unsat[c] \leadsto r \in alloc[c]$
$InfOftenSatisfied \triangleq$ entire request of each client will eventually be satisfied
$\quad \forall c \in Clients : \Box\Diamond(unsat[c] = \{\})$

Figure 2: Correctness properties of the resource allocator.

asserts that the sets of resources allocated to different clients are disjoint; it should be an invariant of the system. The remaining formulas express liveness properties. Formula *ClientsWillReturn* states that whenever some client $c$ has no pending resources then eventually it will not hold any resources. In particular, clients whose request has been entirely satisfied will eventually return the resources that have been allocated: this is precisely requirement (4) of the informal description. Formula *ClientsWillObtain* asserts that whenever client $c$ has requested resource $r$, it will eventually obtain it; this is the fundamental liveness property formulated in section 1. Finally, formula *InfOftenSatisfied* expresses that for every client, the set of its pending requests is infinitely often empty.

Before engaging on a full-scale verification effort, it is a good idea to analyze models using TLC, the TLA$^+$ model checker, which can analyze the state space of finite-state instances of TLA$^+$ models. Besides the TLA$^+$ model written in an ASCII representation, TLC requires a second input file, called the *configuration file*, that defines the finite-state instance to analyze, and that declares the specification and the properties to verify. (A TLA$^+$ model essentially consists of a list of definitions, it does not indicate which of the formulas represents the system specification to analyze. Also, TLC does not interpret any theorems asserted in a module.) Figure 3 shows a configuration file for analyzing module *SimpleAllocator*. It defines a concrete instance of module *SimpleAllocator* by defining sets *Clients* and *Resources* that, in our case, contain symbolic constants. The keyword `SPECIFICATION` indicates the formula representing the main system specification. Finally, the keywords `INVARIANTS` and `PROPERTIES` define the properties to be verified by TLC. This is a very simple example of a configuration

```
CONSTANTS
   Clients = {c1,c2,c3}
   Resources = {r1,r2}

SPECIFICATION
   SimpleAllocator

INVARIANTS
   TypeInvariant ResourceMutex

PROPERTIES
   ClientsWillReturn ClientsWillObtain InfOftenSatisfied
```

Figure 3: Sample configuration file `SimpleAllocator.cfg` for TLC.

file that suffices for our case study; a detailed explanation of configuration files appears in [3] and in the tool documentation available at [2].

When I run TLC on this model on my laptop, I obtain the following output (some details will vary depending on the version and the installation of TLC):

```
TLC Version 2.0 of Mar 17, 2004
Model-checking
Parsing file SimpleAllocator.tla
Parsing file /tools/tla/tlasany/StandardModules/FiniteSets.tla
Parsing file /tools/tla/tlasany/StandardModules/Naturals.tla
Parsing file /tools/tla/tlasany/StandardModules/Sequences.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module FiniteSets
Semantic processing of module SimpleAllocator
Implied-temporal checking--satisfiability problem has 10 branches.
Finished computing initial states: 1 distinct state generated.
--Checking temporal properties for the complete state space...
Model checking completed. No error has been found.
  Estimates of the probability that TLC did not check all reachable states
  because two distinct states had the same fingerprint:
    calculated (optimistic):  2.673642557349254E-14
    based on the actual fingerprints:  6.871173129000332E-15
1633 states generated, 400 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 6.
```

TLC invokes the syntactic and semantic analyzer TLASANY to parse the TLA$^+$ input file and check for well-formedness. It then computes the graph of reachable states for the instance of our model defined by the configuration file, verifying the invariants along the way. Finally, the temporal properties are verified over the state space. In our case, TLC reports that it has not found any error. During the calculation of the state space, TLC compares states based on a hash code ("fingerprint") rather than comparing states precisely, for better efficiency. In case of a hash collision, TLC will mistakenly identify two distinct states and may therefore miss part of the state space. TLC attempts to estimate the probability that this error occurred during the run, based on the distribution of the fingerprints. TLC also reports the number of states it generated during its analysis, the number of distinct states, and the depth of the state graph, i.e. the length of the longest cycle. These statistics can be valuable information: too few states may indicate that some action guards are too strong, while too many states may point to missing conjuncts in guards, resulting in invariant violations. It is a good idea to use TLC to verify every property you can think of, as well as some non-properties: for example, assert the negation of every action guard as an invariant in order to let TLC compute a finite execution that ends in a state where the action can actually be activated. For our example, the TLC run completes after just under 10 seconds (half of that time is spent on the verification of property *ClientsWillObtain*, which is expanded into six properties, for each combination of clients and resources).

After this initial success, we can try to analyze somewhat larger models, but this exploration is limited by the well-known problem of state-space explosion. For example, increasing the number of resources from 2 to 3 in our model results in a state graph that contains 8000 distinct states (among 45697 states generated in all), and the analysis will take roughly 10 minutes instead of 10 seconds (2 minutes when dropping the property *ClientsWillObtain*).

You may notice that the specification and the properties to be verified are invariant with respect to permutations of the sets of clients and resources, and TLC implements a technique of symmetry reduction, which can counteract the effect of state-space explosion: it suffices to extend the TLA$^+$ module by a definition of the predicate

$$Symmetry \triangleq Permutations(Clients) \cup Permutations(Resources)$$

(the operator *Permutations* is defined in the standard TLC module, which must therefore be added to the EXTENDS clause) and to indicate

```
SYMMETRY Symmetry
```

in the configuration file. Unfortunately, the implementation of symmetry reduction in TLC is not compatible with checking liveness properties, and in fact, TLC reports a meaningless "counter-example" when we enable symmetry reduction during the verification of the liveness properties of our example. However, when restricted to checking the invariants, symmetry reduction with respect to both parameter sets reduces the number of states explored to 50 (respectively 309 for three clients and three resources); the runtimes are reduced to fractions of a second for either configuration.

We can easily use TLC to explore variations of our specification. For example, you may want to try to replace the fairness condition for the allocator by one of the following formulas:

$$\begin{array}{ll} (1) & \forall c \in Clients : \mathrm{WF}_{vars}(\exists S \in \mathrm{SUBSET}\, Resources : Allocate(c,S)) \\ (2) & \mathrm{SF}_{vars}(\exists c \in Clients, S \in \mathrm{SUBSET}\, Resources : Allocate(c,S)) \\ (3) & \forall c \in Clients, r \in Resources : \mathrm{SF}_{vars}(Allocate(c,\{r\})) \end{array}$$

Try to understand these conditions and to predict the outcome of TLC before running it on the modified model, then interpret the results!

## 3.2 Analysis by deductive-algorithmic verification

Model checking can give some confidence in the correctness of a model since properties that do not hold of a model usually fail already for small instances. In fact, one way to verify a system is to verify correctness of an instance for small parameter values (say, two or three clients and resources), combined with a proof of data-independence that establishes that the failure of a property to hold over an arbitrary model would already manifest itself over the chosen instance. More traditional is deductive verification of a model, based on proof rules for establishing system properties. This approach is independent of particular instances and can be aided by interactive or automatic theorem provers. The drawback, of course, is the limited degree of automation: the proof has to be planned in minute detail, and the verification rules tend to generate many proof obligations. When assisted by a mechanical theorem prover, one should also have intimate knowledge of its proof methods.

These difficulties can to some extent be alleviated by a hybrid approach that combines theorem proving and model checking, based on the concept of *predicate abstraction*. In this framework, the system is represented as a finite-state transition system whose states are labelled by predicates and thus represent (possibly infinite) sets of system states. The overall verification can be decomposed into two subproblems: on the one hand, we have to prove that the abstraction correctly represents the concrete systems, and this is established using theorem proving, involving no or very little temporal logic. On the other hand, the temporal correctness property of interest is verified

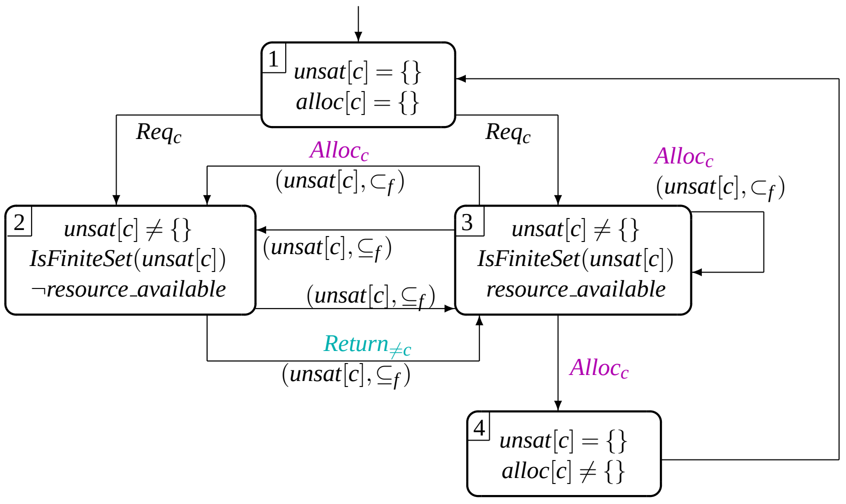

Figure 4: Predicate diagram for the simple allocator.

over the finite-state abstraction by model checking. At our group in Nancy, we have developed a visual representation of predicate abstractions that we call *predicate diagrams* [1], and a supporting tool is currently under development. I will not present predicate diagrams in detail in this note, but only demonstrate the use of the formalism (and, in fact, of a beta version of our tool) to verify that the property *InfOftenSatisfied* holds of specification *SimpleAllocator*.

The predicate diagram shown in Fig. 4 contains 4 nodes numbered 1 to 4. Nodes are labeled with predicates, and edges are associated with actions and, possibly, with ordering annotations. For example, the diagram asserts that action $Req_c$ may take us from the initial node 1 to either node 2 or node 3. Similarly, the action $Alloc_c$ may reach nodes 2, 3 or 4 from node 3, the first two transitions ensuring that $unsat[c]$ decreases with respect to the well-founded ordering $\subset_f$, i.e. implying $unsat[c]' \subset_f unsat[c]$. The predicate diagram is formally based on the TLA$^+$ module shown in Fig. 5, which introduces some abbreviations for predicate and actions.

Informally, it is not hard to see that the predicate diagram faithfully reflects transitions of the system. Besides the explicitly shown transitions, predicate diagrams also allow for transitions that loop at the source node. (If that node has some outgoing edge with an ordering annotation $(t, \prec)$, all looping transitions must ensure that the term $t$ does not increase with respect to that ordering.) For example, assume that the system is in some state represented by node 3, i.e. where client $c$ has some pending request and where some of the requested resources are currently available, and consider the possible system transitions (for some client $d$ and set $S$ of resources):

$Request(d,S)$ : because $unsat[c] \neq \{\}$ holds in the source state, $d$ is different from $c$. Therefore, the transition changes neither $unsat[c]$ nor *available*, and it is represented by the implicit looping transition from node 3.

$Return(d,S)$ : such transitions do not change $unsat[c]$ (even if $d = c$), and they ensure $available' \supseteq available$, so again they preserve the predicates that characterize node 3.

$Allocate(d,S)$ : we consider several subcases:

MODULE *SimpleAllocatorDiagram*

EXTENDS *SimpleAllocator*
CONSTANT $c$
ASSUME $c \in Clients$

$resource\_available \triangleq unsat[c] \cap available \neq \{\}$
$Req_c \triangleq \exists S \in \text{SUBSET}\ Resources : Request(c, S)$
$Alloc_c \triangleq \exists S \in \text{SUBSET}\ Resources : Allocate(c, S)$
$Return_{\neq c} \triangleq$
$\quad \wedge\ unsat[c] \neq \{\} \wedge \neg resource\_available$
$\quad \wedge\ \exists d \in Clients : d \neq c \wedge alloc[d] \cap unsat[c] \neq \{\} \wedge Return(d, alloc[d])$
$S \subset_f T \triangleq IsFiniteSet(S) \wedge S \subseteq T \wedge S \neq T$
$S \subseteq_f T \triangleq IsFiniteSet(S) \wedge S \subseteq T$

Figure 5: Auxiliary definitions for the predicate diagram.

- If $d = c$ and $unsat[c]' = \{\}$ (i.e., when $S = unsat[c]$) then clearly we have $alloc[c]' \neq \{\}$, and the transition is therefore represented by the edge from node 3 to node 4.
- Otherwise, if $d = c$, we have $\{\} \neq unsat[c]' \subset_f unsat[c]$, and the predicate *resource_available* may or may not hold in the post-state. The transition is therefore represented either by the explicit loop at node 3 or by the edge labeled $Alloc_c$ from node 3 to node 2.
- Finally, if $d \neq c$, the set $unsat[c]$ stays unchanged, and *resource_available* may or may not hold in the post-state. The transition is therefore represented either by the implicit loop at node 3 (which, as mentioned before, is annotated by $(unsat[c], \subseteq_f)$), or by the lower edge from node 3 to node 2: edges without an explicitly given action name are associated with the system's next-state relation.

Similar reasoning justifies the transitions for the remaining diagram nodes. Formally, the correctness of the diagram is established by a set of proof obligations that have been defined in [1] and that are generated (but not yet proven) by our tool. Our goal is to verify the property

$$\Box\Diamond(unsat[c] = \{\})$$

(from which property *InfOftenSatisfied* follows by the $\forall$-introduction rule of ordinary predicate logic). To do so, we must show that every path passes infinitely often through nodes 1 or 4. At first sight, this is not obvious: the system may loop at or between nodes 2 and 3. In fact, the various edge annotations serve to show that these loops must eventually be exited. First, we assume weak fairness for the action $Return_{\neq c}$, and this ensures that node 2 will eventually be left (it is easy to see that the predicate label of node 2 ensures that the action is enabled). Thus, any trace looping at nodes 2 and 3 must visit node 3 infinitely often. Secondly, we assume strong fairness for the action $Alloc_c$; clearly, this follows immediately from the fairness assumption of specification *SimpleAllocator*. Because $Alloc_c$ is enabled whenever the trace visits node 3, the action must be taken infinitely often along our assumed loop. However, each such transition implies that $unsat[c]' \subset_f unsat[c]$, whereas no other transition along the loop increases

$unsat[c]$. Therefore, we obtain an infinite descent with respect to a well-founded ordering, which is of course impossible, and the loop must eventually be left by visiting node 4.

This informal reasoning is formally backed up by our tool by starting a model checker to verify the property over a finite-state model generated from the diagram, and this can be very helpful to determine the necessary annotations. For example, I had at first omitted the annotations $(unsat[c], \subseteq_f)$ for the edges between nodes 2 and 3, except for the edge for the $Alloc_c$ action. The model checker immediately produced a counter example that pointed to the missing annotations.

Of course, the verification of properties from predicate diagrams is conclusive only if we can actually discharge all the proof obligations that ensure the correctness of the diagram with respect to the underlying specification. Most of them are non-temporal formulas, concerning the enabledness or the effect of transitions at certain system states. However, we must also show that the system specification implies the fairness conditions associated with the diagram. These verification conditions are often obvious, however, in the example above we must prove

$$SimpleAllocator \Rightarrow \mathrm{WF}_{vars}(Return_{\neq c})$$

whose correctness requires some reasoning about the interaction of quantifiers and fairness conditions (see [3, ch. 8.5.3]).

Finally, one may observe that the transition from node 4 to node 1 of the diagram corresponds to an occurrence of the action $Return(c, alloc[c])$, for which we have assumed weak fairness. We can add this information to our predicate diagram and use it to verify the property

$$\Box\Diamond(alloc[c] = \{\})$$

proving that client $c$ (and, therefore, all clients) will eventually return their resources.

## 3.3 The specification revisited

We have analyzed the *SimpleAllocator* specification using both model checking and theorem proving techniques and have successfully verified its correctness properties. Does this mean that the specification is correct? Consider the following scenario: two clients $c_1$ and $c_2$ both request resources $r_1$ and $r_2$. The allocator grants $r_1$ to $c_1$ and $r_2$ to $c_2$. From our informal description in section 1, neither client should be required to give up any of its resources before it has received all resources it requested. On the other hand, neither client can acquire the entire set of resources it requested while the other client is holding its resource, so the model appears to be deadlocked. Why didn't TLC report any deadlock in the model?

The reason is of course that, technically, the model is not deadlocked: both clients have an enabled action because they *may* return the resource they are holding according to requirement (3) of the problem description. The problem is that our model actually *requires* clients to eventually return the resources they are holding, via the fairness assertion

$$\forall c \in Clients : \mathrm{WF}_{vars}(Return(c, alloc[c]))$$

whereas the informal requirement (4) only asks that client return their resources provided they have no outstanding requests. We should therefore have imposed the weaker fairness condition

$$\forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}\big(\mathit{unsat}[c] = \{\} \land \mathit{Return}(c, \mathit{alloc}[c])\big)$$

Rerunning TLC on the modified specification produces the following output (after the initial diagnostic messages):

```
Error: Temporal properties were violated.
The following behaviour constitutes a counter-example:

STATE 1: <Initial predicate>
/\ unsat = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {} @@ c2 :> {} @@ c3 :> {})

STATE 2: <Action line 43, col 3 to line 45, col 50 ...>
/\ unsat = (c1 :> {r1, r2} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {} @@ c2 :> {} @@ c3 :> {})

STATE 3: <Action line 50, col 3 to line 52, col 50 ...>
/\ unsat = (c1 :> {r2} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})

STATE 4: <Action line 43, col 3 to line 45, col 50 ...>
/\ unsat = (c1 :> {r2} @@ c2 :> {r1, r2} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})

STATE 5: <Action line 50, col 3 to line 52, col 50 ...>
/\ unsat = (c1 :> {r2} @@ c2 :> {r1} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {r2} @@ c3 :> {})

STATE 6: <Action line 43, col 3 to line 45, col 50 ...>
/\ unsat = (c1 :> {r2} @@ c2 :> {r1} @@ c3 :> {r1, r2})
/\ alloc = (c1 :> {r1} @@ c2 :> {r2} @@ c3 :> {})

STATE 7: Back to state 6.
```

In fact, TLC produces (essentially) the counter-example described above: clients c1 and c2 request both available resources, the allocator grants each client access to one resource, and no more progress is made from there since the trace ends in infinite stuttering. As a result, the trace violates the liveness properties *ClientsWillObtain* and *InfOftenSatisfied* (TLC does not indicate precisely which properties are not satisfied by the counterexample).

The moral of this observation is that models can be inappropriate even if they satisfy all correctness properties, and that validation of models is extremely important before engaging on verification. Of course, any experienced user of TLA$^+$ would have pointed out the problem immediately, but similar problems may be much harder to find in real-world specifications. Stepwise development by refinement can help to some extent because high-level specifications are smaller. However, it may also become more difficult to express adequate liveness assumptions at a higher level of abstraction.

## 4 A scheduling allocator

Specification *SimpleAllocator* is too simple because the allocator is free to allocate resources in any order. Therefore, it may “paint itself into a corner” as in the run discussed in the previous section, requiring cooperation from the clients to recover. We can prevent this from happening by having the allocator fix a schedule according to which access to resources will be granted. We present a formal TLA$^+$ model based on this idea and use TLC to analyze it.

### 4.1 TLA$^+$ model

A specification of a “scheduling” allocator appears in module *SchedulingAllocator* in Fig. 6. It is based on three state variables: as before, the variables *unsat* and *alloc* indicate the sets of pending requests and allocated resources per client. The variable *sched* contains a sequence of clients that represents the schedule according to which resources will be granted.

The module contains a number of auxiliary definitions: again we find a typing invariant that asserts the types of the state variables, and the set *available* of resources that are available for allocation. The operator $Range(f)$ computes the range of a function; because sequences are functions in TLA$^+$, it can also be used to compute the elements that appear in a sequence. The set *toSchedule* is defined to contain the clients with unsatisfied requests that do not appear in the schedule. The operator *Drop* is defined such that $Drop(seq, i)$ returns the sequence *seq* from which the *i*-th element has been removed. Somewhat more involved is the definition of $PermSeqs(S)$ that computes the set of permutation sequences of a finite set $S$. The idea is that $\langle x_1, \ldots, x_n \rangle$ is a permutation of a non-empty finite set $S$ if and only if $\langle x_1, \ldots, x_{n-1} \rangle$ is a permutation of $S \setminus \{x_n\}$. The formal expression in TLA$^+$ makes use of an auxiliary, recursively defined, function *perms* that computes the set of permutations $perms[T]$ of any subset $T \subseteq S$, in a style that is similar to the recursive definition of functions over inductive data types in a functional programming language.

The initial state predicate *Init* should be obvious: we again assume that we start in a state where there are no outstanding requests and where no resources have been allocated; also, the schedule is empty.

The definitions of the actions $Request(c, S)$, $Allocate(c, S)$, and $Return(c, S)$ extend the corresponding definitions of the simple allocator by updating the variable *sched*. In particular, the action $Allocate(c, S)$ modeling allocation of the resources in $S$ to client $c$ adds a constraint to the corresponding action of the simple allocator: client $c$ must have been scheduled for allocation, and no client appearing earlier in the schedule must have an outstanding request for any resource in $S$. Moreover, if $S$ equals the set of outstanding requests of client $c$ (and therefore $c$’s request is completely satisfied after the allocation of $S$), $c$ is dropped from the schedule.

We introduce a new *Schedule* action that models the scheduling of clients with outstanding requests by the allocator. It is enabled iff the set *toSchedule* is non-empty; its effect is to extend the current schedule by some arbitrary permutation of set *toSchedule*. Of course, a real allocator would prescribe some strategy according to which waiting clients are scheduled, and this would correspond to a refinement of the *Schedule* action described here. The purpose of this note is not to suggest some specific such strategy, but to verify the correctness of the protocol for any strategy.

The specification of the allocator appears as formula *Allocator* at the bottom of the module. The safety part should be obvious. The two first fairness assumptions are

```
MODULE $SchedulingAllocator$
EXTENDS $FiniteSet, Sequences, Naturals$
CONSTANTS $Clients, Resources$
ASSUME $IsFiniteSet(Resources)$
VARIABLES $unsat, alloc, sched$

$TypeInvariant \triangleq$
    $\land\ unsat \in [Clients \rightarrow \text{SUBSET}\ Resources]\ \land\ alloc \in [Clients \rightarrow \text{SUBSET}\ Resources]$
    $\land\ sched \in Seq(Clients)$
$available \triangleq Resources \setminus (\text{UNION}\ \{alloc[c] : c \in Clients\})$
$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN}\ f\}$
$toSchedule \triangleq \{c \in Clients : unsat[c] \neq \{\} \land c \notin Range(sched)\}$
$PermSeqs(S) \triangleq$          set of permutations of finite set S, represented as sequences
    LET $perms[ss \in \text{SUBSET}\ S] \triangleq$
            IF $ss = \{\}$ THEN $\langle\rangle$
            ELSE LET $ps \triangleq [x \in ss \mapsto \{Append(sq, x) : sq \in perms[ss \setminus \{x\}]\}]$
                 IN   UNION $\{ps[x] : x \in ss\}$
    IN   $perms[S]$
$Drop(seq, i) \triangleq SubSeq(seq, 1, i-1) \circ SubSeq(seq, i+1, Len(seq))$

$Init \triangleq unsat = [c \in Clients \mapsto \{\}] \land alloc = [c \in Clients \mapsto \{\}] \land sched = \langle\rangle$
$Request(c, S) \triangleq$
    $\land\ unsat[c] = \{\} \land alloc[c] = \{\} \land S \neq \{\}$
    $\land\ unsat' = [unsat\ \text{EXCEPT}\ ![c] = S] \land \text{UNCHANGED}\ \langle alloc, sched\rangle$
$Allocate(c, S) \triangleq$
    $\land\ S \neq \{\} \land S \subseteq available \cap unsat[c]$
    $\land\ \exists i \in \text{DOMAIN}\ sched :$
          $\land\ sched[i] = c \land \forall j \in 1..i-1 : unsat[sched[j]] \cap S = \{\}$
          $\land\ sched' = \text{IF}\ S = unsat[c]\ \text{THEN}\ Drop(sched, i)\ \text{ELSE}\ sched$
    $\land\ alloc' = [alloc\ \text{EXCEPT}\ ![c] = @ \cup S]$
    $\land\ unsat' = [unsat\ \text{EXCEPT}\ ![c] = @ \setminus S]$
$Return(c, S) \triangleq$
    $\land\ S \neq \{\} \land S \subseteq alloc[c]$
    $\land\ alloc' = [alloc\ \text{EXCEPT}\ ![c] = @ \setminus S] \land \text{UNCHANGED}\ \langle unsat, sched\rangle$
$Schedule \triangleq$
    $\land\ toSchedule \neq \{\}$
    $\land\ \exists sq \in PermSeqs(toSchedule) : sched' = sched \circ sq$
    $\land\ \text{UNCHANGED}\ \langle unsat, alloc\rangle$
$Next \triangleq$
    $\lor\ \exists c \in Clients, S \in \text{SUBSET}\ Resources : Request(c, S) \lor Allocate(c, S) \lor Return(c, S)$
    $\lor\ Schedule$
$vars \triangleq \langle unsat, alloc, sched\rangle$

$Allocator \triangleq \land\ Init \land \Box[Next]_{vars}$
                $\land\ \forall c \in Clients : \text{WF}_{vars}(unsat[c] = \{\} \land Return(c, alloc[c]))$
                $\land\ \forall c \in Clients : \text{WF}_{vars}(\exists S \in \text{SUBSET}\ Resources : Allocate(c, S))$
                $\land\ \text{WF}_{vars}(Schedule)$
```

Figure 6: Specification of an allocator with scheduling.

$$
\begin{array}{l}
\mathit{UnscheduledClients} \triangleq \qquad \text{set of clients that are not in the schedule} \\
\quad \mathit{Clients} \setminus \mathit{Range}(\mathit{sched}) \\
\mathit{PrevResources}(i) \triangleq \qquad \text{available resources when } i\text{th process has to be satisfied} \\
\qquad \mathit{available} \\
\quad \cup\ \textsc{union}\ \{\mathit{unsat}[\mathit{sched}[j]] \cup \mathit{alloc}[\mathit{sched}[j]] : j \in 1..i-1\} \\
\quad \cup\ \textsc{union}\ \{\mathit{alloc}[c] : c \in \mathit{UnscheduledClients}\} \\
\mathit{AllocatorInvariant} \triangleq \\
\quad \land\ \forall c \in \mathit{toSchedule} : \mathit{unsat}[c] \neq \{\} \ \land\ \mathit{alloc}[c] = \{\} \\
\quad \land\ \forall i \in \textsc{domain}\ \mathit{sched} : \begin{array}[t]{l} \land\ \mathit{unsat}[\mathit{sched}[i]] \neq \{\} \\ \land\ \forall j \in 1..i-1 : \mathit{alloc}[\mathit{sched}[i]] \cap \mathit{unsat}[\mathit{sched}[j]] = \{\} \\ \land\ \mathit{unsat}[\mathit{sched}[i]] \subseteq \mathit{PrevResources}(i) \end{array}
\end{array}
$$

Figure 7: Lower-level invariant of scheduling allocator.

similar to those of the *SimpleAllocator* module: we impose the (corrected) fairness assumption for returning resources discussed in section 3.3. For the allocation action, we require the scheduler to be fair with respect to all clients for which allocation is possible. Some thinking should convince you that it is now enough to require weak fairness instead of the strong fairness condition imposed on the simple allocator because clients acquire priority according to their rank in the schedule. A new fairness condition is asserted of the *Schedule* action such that the allocator will eventually schedule clients that need to be scheduled.

## 4.2 Analysis using TLC

We can again use TLC to verify the correctness properties described in section 3.1. Again, the typing invariant, the exclusive access to resources, and the three liveness properties are verified over a sample model consisting of three clients and two resources. TLC computes 1690 distinct states (out of 5854 states generated), requiring roughly 39 seconds for the analysis (the run time drops to 13 seconds when the property *ClientsWillObtain* is omitted). What sets TLC apart from more conventional model checkers is its ability to analyze the model at the high level of abstraction at which it has been presented in Fig. 6: neither the definition of the operator *PermSeqs* nor the relatively complicated fairness constraints pose a problem. (For better efficiency, we could override the definition of *PermSeqs* by a method written in Java, but this is not a big concern for a list that contains at most three elements.)

Given the experience with the verification of the simple allocator model, one should be suspicious of the quick success with the new model. As Lamport [3, ch. 14.5.3] writes, it is a good idea to verify as many properties as possible. Figure 7 contains a lower-level invariant of the scheduling allocator that can be verified using TLC. The first conjunct says that all clients in set *toschedule* have a non-empty set of outstanding resources, but hold no resources. The second conjunct concerns the clients in the schedule; it is split into three sub-conjuncts: first, each client in the schedule has some outstanding requests, second, no client may hold some resource that is requested by some prioritized client (appearing earlier in the schedule), and finally, the set of outstanding requests of a client in the schedule is bounded by the union of the set of currently available resources, the resources requested or held by prioritized clients and the resources held by clients that do not appear in the schedule. The idea behind this last conjunct is to assert that a client's requests can be satisfied using resources that

```
---------------- MODULE AllocatorRefinement ----------------
EXTENDS SchedulingAllocator
Simple ≜ INSTANCE SimpleAllocator
SimpleAllocator ≜ Simple!SimpleAllocator
-------------------------------------------------------------
THEOREM Allocator ⇒ SimpleAllocator
=============================================================
```

Figure 8: A module asserting refinement of the allocator specifications.

are either already free or that are held by prioritized clients. It follows that prioritized clients can obtain their full set of resources, after which they are required to eventually release them again. Therefore, the scheduling allocator works correctly even under the worst-case assumption that clients will only give up resources after their complete request has been satisfied.

Beyond these correctness properties, TLC can also establish a formal refinement relationship between the two specifications. In fact, the specification of the scheduling allocator adds constraints on the allocation of resources. It also specifies the behavior of the variable *sched*, which did not appear in the specification of the simple allocator, and which is therefore not constrained by that specification. More interestingly, the scheduling policy and the (weaker) liveness assumptions imply that the (original) fairness constraints are effectively met.

In order to check refinement by TLC, we define the module *AllocatorRefinement* that appears in Fig. 8. It extends module *SchedulingAllocator*, thus importing all declarations and definitions of that module, and defines an instance *Simple* of module *SimpleAllocator*, whose parameters are (implicitly) instantiated by the entities of the same name inherited from module *SchedulingAllocator*. All operators *Op* defined in the instance are available as *Simple*!*Op*. (It would have been illegal to extend both modules *SchedulingAllocator* and *SimpleAllocator* because they declare constants and variables, as well as define operators, of the same names.) The module then asserts refinement of specification *SimpleAllocator* by specification *Allocator*; remember that refinement is just implication in TLA$^+$. We could of course have written

THEOREM *Allocator* ⇒ *Simple*!*SimpleAllocator*

in the formulation of the theorem; to make the statement more readable we have defined the formula *SimpleAllocator* to stand for *Simple*!*SimpleAllocator*. In fact, this auxiliary definition is also required for analysis with TLC, whose present version does not allow us to specify formulas of the form *Module*!*Property* in the configuration file. We can call on TLC to verify the implication over the instance of the model consisting of three clients and two resources just by asserting

```
PROPERTIES SimpleAllocator
```

in the configuration file. TLC declares the implication to be valid, for our small instance of three clients and two resources in approximately 6 seconds. Since we already knew from section 3.1 that the (original) simple allocator specification satisfied the correctness properties, it would not actually have been necessary to re-verify them for the scheduling allocator: they are guaranteed to hold by transitivity of implication!

## 5 Towards an implementation

The specification of module *SchedulingAllocator* describes an overall algorithm (or rather a class of algorithms) for resource allocation; analysis by TLC has indicated that this algorithm satisfies all desired correctness properties, even under worst-case assumptions about the clients' behavior. Our next goal is to refine that specification into one that is implementable as a distributed system. In particular, we will assume that the allocator and the clients may run on different computers. Therefore, each process should have direct access only to its local memory. It must use explicit communication by message passing to interact with other processes. Instead of a centralized representation of the system state based on the variables *unsat* and *alloc*, we will have to distinguish between the allocator's view and each client's view of its pending requests and allocated resources. Similarly, the basic actions such as the request for resources will be split into two parts, with different processes being responsible for carrying them out: in a first step, the client issues a request, updates its local state, and sends a corresponding message to the allocator. Subsequently, the allocator receives the message and updates its table of pending requests accordingly.

Figures 9 and 10 contain a TLA$^+$ model based on this idea. It contains variables *unsat*, *alloc*, and *sched* as before, but we now consider these to be local variables of the allocator. New variables *requests* and *holding* represent the clients' views of pending resource requests and of resources currently held; we interpret $requests[c]$ and $holding[c]$ as being local to the client process $c$. The variable *network* represents the set of messages in transit between the allocator and the clients.

Except for action *Schedule*, which is a private action of the allocator, all actions that appeared in specification *SchedulingAllocator* have been split into two actions as explained above. For example, client $c$ is considered to perform action $Request(c,S)$ because only its "local" variables and the state of the communication network are modified by the action. The allocator later performs action $RReq(m)$, for a suitable value of $m$. The fairness conditions of our previous specification are complemented by weak fairness requirements for the actions $RReq(m)$, $RAlloc(m)$, and $RRet(m)$, that are associated with message reception (for all possible messages $m$); these requirements express that messages will eventually be received and handled.

A remark is in order concerning the style in which module *AllocatorImplementation* is written: I describe it as representing a distributed system, attributing state components and actions to different processes. Formally, however, this (de-)composition is not reflected in the structure of the TLA$^+$ formula, which is again of the form

$$Init \land \Box[Next]_{vars} \land Liveness.$$

I could instead have structured the specification as a composition (conjunction) of separate process specifications, and Lamport discusses this issue in [3, ch. 10]. Monolithic specifications are usually easier to write. Moreover, the current version of TLC does not support compositions of processes with separate next-state relations and fairness assumptions. One can debate whether the style I have chosen is adequate or not, and in fact I believe that there is room for research on how to make it more practical to write and analyze distributed systems written as a composition of individual processes. Whether there is much practical merit in doing so is less clear to me at this moment, and a detailed discussion is certainly beyond the scope of this tutorial note, so I'll return to the technical discussion of the specification as it stands.

Module *AllocatorImplementation* claims that the model obtained in this way is a refinement of the scheduling allocator specification, and we can again use TLC to ver-

```
┌──────────────────── MODULE AllocatorImplementation ────────────────────┐
EXTENDS FiniteSets, Sequences, Naturals
CONSTANTS Clients, Resources
ASSUME IsFiniteSet(Resources)
VARIABLES unsat, alloc, sched, requests, holding, network
Sched ≜ INSTANCE SchedulingAllocator
├─────────────────────────────────────────────────────────────────────────┤
Messages ≜
   [type : {"request", "allocate", "return"}, clt : Clients, rsrc : SUBSET Resources]
TypeInvariant ≜
   ∧ Sched!TypeInvariant
   ∧ requests ∈ [Clients → SUBSET Resources]
   ∧ holding ∈ [Clients → SUBSET Resources]
   ∧ network ∈ SUBSET Messages
├─────────────────────────────────────────────────────────────────────────┤
Init ≜
   ∧ Sched!Init
   ∧ requests = [c ∈ Clients ↦ {}] ∧ holding = [c ∈ Clients ↦ {}] ∧ network = {}
Request(c, S) ≜          client c requests set S of resources
   ∧ requests[c] = {} ∧ holding[c] = {} ∧ S ≠ {}
   ∧ requests′ = [requests EXCEPT ![c] = S]
   ∧ network′ = network ∪ {[type ↦ "request", clt ↦ c, rsrc ↦ S]}
   ∧ UNCHANGED ⟨unsat, alloc, sched, holding⟩
RReq(m) ≜          allocator handles request message sent by some client
   ∧ m ∈ network ∧ m.type = "request" ∧ network′ = network \ {m}
   ∧ unsat′ = [unsat EXCEPT ![m.clt] = m.rsrc]
   ∧ UNCHANGED ⟨alloc, sched, requests, holding⟩
Allocate(c, S) ≜          allocator decides to allocate resources S to client c
   ∧ Sched!Allocate(c, S)
   ∧ network′ = network ∪ {[type ↦ "allocate", clt ↦ c, rsrc ↦ S]}
   ∧ UNCHANGED ⟨requests, holding⟩
RAlloc(m) ≜          some client receives resource allocation message
   ∧ m ∈ network ∧ m.type = "allocate" ∧ network′ = network \ {m}
   ∧ holding′ = [holding EXCEPT ![m.clt] = @ ∪ m.rsrc]
   ∧ requests′ = [requests EXCEPT ![m.clt] = @ \ m.rsrc]
   ∧ UNCHANGED ⟨unsat, alloc, sched⟩
Return(c, S) ≜          client c returns resources in S
   ∧ S ≠ {} ∧ S ⊆ holding[c]
   ∧ holding′ = [holding EXCEPT ![c] = @ \ S]
   ∧ network′ = network ∪ {[type ↦ "return", clt ↦ c, rsrc ↦ S]}
   ∧ UNCHANGED ⟨unsat, alloc, sched, requests⟩
RRet(m) ≜          allocator receives returned resources
   ∧ m ∈ network ∧ m.type = "return" ∧ network′ = network \ {m}
   ∧ alloc′ = [alloc EXCEPT ![m.clt] = @ \ m.rsrc]
   ∧ UNCHANGED ⟨unsat, sched, requests, holding⟩
Schedule ≜ Sched!Schedule ∧ UNCHANGED ⟨requests, holding, network⟩
├─────────────────────────────────────────────────────────────────────────┤
```

Figure 9: An implementation of the allocator (part 1).

$$
\begin{array}{l}
\mathit{Next} \;\triangleq \\
\quad \lor\ \exists c \in \mathit{Clients}, S \in \textsc{subset}\ \mathit{Resources} : \mathit{Request}(c,S) \lor \mathit{Allocate}(c,S) \lor \mathit{Return}(c,S) \\
\quad \lor\ \exists m \in \mathit{network} : \mathit{RReq}(m) \lor \mathit{RAlloc}(m) \lor \mathit{RRet}(m) \\
\quad \lor\ \mathit{Schedule} \\
\mathit{vars} \;\triangleq\; \langle \mathit{unsat}, \mathit{alloc}, \mathit{sched}, \mathit{requests}, \mathit{holding}, \mathit{network} \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{Liveness} \;\triangleq \\
\quad \land\ \forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\mathit{requests}[c] = \{\} \land \mathit{Return}(c, \mathit{holding}[c])) \\
\quad \land\ \forall c \in \mathit{Clients} : \mathrm{WF}_{\mathit{vars}}(\exists S \in \textsc{subset}\ \mathit{Resources} : \mathit{Allocate}(c,S)) \\
\quad \land\ \mathrm{WF}_{\mathit{vars}}(\mathit{Schedule}) \\
\quad \land\ \forall m \in \mathit{Messages} : \mathrm{WF}_{\mathit{vars}}(\mathit{RReq}(m)) \land \mathrm{WF}_{\mathit{vars}}(\mathit{RAlloc}(m)) \land \mathrm{WF}_{\mathit{vars}}(\mathit{RRet}(m)) \\
\mathit{Implementation} \;\triangleq\; \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathit{Liveness}
\end{array}
$$

$$
\textsc{theorem}\ \mathit{Implementation} \Rightarrow \mathit{Sched}!\mathit{Allocator}
$$

Figure 10: An implementation of the allocator (part 2).

ify this theorem for our usual instance of three client processes and two resources. However, TLC quickly produces a counterexample that ends in the following states:

```
STATE 7:
/\ holding = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ requests = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ sched = << >>
/\ network = {[type |-> "return", clt |-> c1, rsrc |-> {r1}]}
/\ unsat = (c1 :> {} @@ c2 :> {} @@ c3 :> {})

STATE 8:
/\ holding = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ requests = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ sched = << >>
/\ network = { [type |-> "request", clt |-> c1, rsrc |-> {r1}],
  [type |-> "return", clt |-> c1, rsrc |-> {r1}] }
/\ unsat = (c1 :> {} @@ c2 :> {} @@ c3 :> {})

STATE 9:
/\ holding = (c1 :> {} @@ c2 :> {} @@ c3 :> {})
/\ alloc = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ requests = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
/\ sched = << >>
/\ network = {[type |-> "return", clt |-> c1, rsrc |-> {r1}]}
/\ unsat = (c1 :> {r1} @@ c2 :> {} @@ c3 :> {})
```

We can infer (cf. state 7) that client `c1` has returned resource `r1` to the allocator. In the transition to state 8, it issues a new request for the same resource, which is handled by the allocator (according to action *RReq*) in the transition to state 9. This action modifies the variable *unsat* at position `c1` although the value of *alloc*[`c1`] is not the empty set—a transition that is not allowed by the scheduling allocator.

$$
\begin{array}{l}
\mathit{RequestsInTransit}(c) \triangleq \qquad \text{requests sent by c but not yet received} \\
\quad \{msg.rsrc : msg \in \{m \in network : m.type = \text{“request”} \land m.clt = c\}\} \\
\mathit{AllocsInTransit}(c) \triangleq \qquad \text{allocations sent to c but not yet received} \\
\quad \{msg.rsrc : msg \in \{m \in network : m.type = \text{“allocate”} \land m.clt = c\}\} \\
\mathit{ReturnsInTransit}(c) \triangleq \qquad \text{return messages sent by c but not yet received} \\
\quad \{msg.rsrc : msg \in \{m \in network : m.type = \text{“return”} \land m.clt = c\}\} \\
\mathit{Invariant} \triangleq \forall c \in \mathit{Clients} : \\
\quad \land\ \mathit{Cardinality}(\mathit{RequestsInTransit}(c)) \leq 1 \\
\quad \land\ \mathit{requests}[c] = \quad \mathit{unsat}[c] \\
\qquad\qquad\qquad\quad \cup\ \textsc{union}\ \mathit{RequestsInTransit}(c) \\
\qquad\qquad\qquad\quad \cup\ \textsc{union}\ \mathit{AllocsInTransit}(c) \\
\quad \land\ \mathit{alloc}[c] = \quad \mathit{holding}[c] \\
\qquad\qquad\qquad \cup\ \textsc{union}\ \mathit{AllocsInTransit}(c) \\
\qquad\qquad\qquad \cup\ \textsc{union}\ \mathit{ReturnsInTransit}(c)
\end{array}
$$

Figure 11: Relating the allocator and client variables by an invariant.

Intuitively, the problem is due to the asynchronous communication network underlying our model, which makes the allocator receive and handle the request message before it receives the earlier return message. In our case, one can argue that the violation of refinement is harmless: a subsequent treatment of the pending return message restores the situation to normal. However, such race conditions are in general better avoided, and there are several ways to do so. For example, an implementation might use FIFO communication between any pair of processes, and the network should then be modeled as a queue (or a set of queues) of messages instead of a set. A less intrusive change is to add the precondition

$$\mathit{alloc}[m.clt] = \{\}$$

to the definition of the action $\mathit{RReq}(m)$, which obviously excludes the run shown above and is also implementable in terms of the local knowledge of the allocator. After this correction, TLC confirms the refinement theorem for our small instance in about 2 minutes. We can also re-verify correctness properties similar to those of section 3.1, but now expressed in terms the clients' variables, such as

$$\forall c \in \mathit{Clients}, r \in \mathit{Resources} : r \in \mathit{requests}[c] \leadsto r \in \mathit{holding}[c]$$

Finally, we can assert the invariant shown in Fig. 11 to relate the variables associated with the clients and the allocator. The verification of these properties for our standard instance of the model generates 64414 states (17701 of which are distinct) and takes roughly 19 minutes (5 minutes without the property *ClientsWillObtain*, 12 seconds for verifying only the invariant properties).

## 6 Some lessons

Given the informal requirements outlined in section 1, newcomers to formal specification and to TLA$^+$ would perhaps have started to write a model similar to the final one presented in this note, or even a more detailed one. It is, however, best to start with as abstract a specification as one can imagine. A low-level specification is at least as likely to contain errors as a program, and the whole purpose of modeling is to clarify and analyze a system at an adequate level of abstraction. The seemingly trivial *SimpleAllocator* specification of Fig. 1 was important because it helped us discover the need for fixing a schedule for resource allocation. It is not clear whether we would have noticed the problem at the level of detail of the final specification, where there are additional problems of synchronization and message passing to worry about. The specification *SchedulingAllocator* corrected the problem discovered for the initial model while staying at the same level of abstraction; it helped us to convince ourselves of the adequacy of the solution. Finally, module *AllocatorImplementation* introduced a step towards a possible implementation by attributing the state variables and the actions to separate processes, and by introducing explicit communication.

For each model, TLC was of great help in analyzing various properties. Although only small instances can be handled by model checking before running into the state explosion problem, doing so greatly increases the confidence in the models. Because TLC is fully automatic, variants of the specifications can be checked without great effort, and various properties (invariants and more general temporal properties) can be verified in a single run. Deductive verification can establish system properties in a fully rigorous way, and I have presented a somewhat more usable approach based on predicate diagrams in section 3.2. However, the price to pay is that the proof has to be planned much more carefully.

Throughout this note, I have emphasized refinement relationships between the different models. Developing a model by successive refinements greatly helps to make formal development manageable; its benefit would become even clearer on a larger example. But even here, we were able to reuse a large part of the *SchedulingAllocator* specification when writing the implementation model, allowing us to focus on decomposition of state and actions and on communication rather than on the underlying allocation algorithm itself. Because refinement preserves properties established at the abstract level, it can even help to reduce the effort needed for verification. For example, TLC verified the final refinement theorem in just 2 minutes, whereas it took 18 minutes to verify the high-level correctness properties over the implementation model.

The example presented in this note illustrates some of the aspects that are central to TLA$^+$, but Lamport's book has of course much more to say. I hope that some readers will find it useful, and I would be happy to receive feedback and suggestions.

## References

[1] D. Cansell, D. Méry, and S. Merz. Diagram refinements for the design of reactive systems. *Journal of Universal Computer Science*, 7(2):159–174, 2001.

[2] L. Lamport. The TLA home page. http://www.research.microsoft.com/users/lamport/tla/tla.html.

[3] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.

[4] L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, 1999.