Liveness Conditions in Model-Based Service Specifications: A Case Study

Alan Fekete,
Department of Computer Science,
University of Sydney, Australia
fekete@cs.su.oz.au

Abstract

Many different formal methods provide mathematical models of reactive systems (those that interact with an environment to provide a service). There is disagreement on many details, but many methods use the transition system as the fundamental representation of a single component; there is also broad agreement that one should specify a service as a whole by presenting a single system that reflects the functional requirements at a high level of abstraction (that is, one gives a global model as specification for a service, even though the actual implementation may be distributed). Many researchers have concentrated on capturing the finite behavior of the specification, but for reasoning about a whole application one needs liveness guarantees on each service. Temporal logic is often suggested as a language for expressing liveness properties, but some formal methods use a different technique based on fairness in the transition system. This paper explores the ease of expressing liveness conditions in a model-based specification. It uses an example from the literature (a cancellable resource allocator) and shows how to express a range of liveness requirements, using both temporal logic and transition fairness. The paper also illustrates common techniques that are used in writing specifications based on transition fairness.

1 Introduction

1.1 Background on Service Specification

Recent years have seen a trend to "open systems" and "component-based programming". In these frameworks, an application is not designed and coded as a monolithic whole, but rather is the combination of several pieces of software, some of which are provided by different vendors. Each separate software entity may indeed be distributed, and therefore constructed from multiple local components; but at a suitable level of abstraction the entity can be seen as a *reactive* system, whose purpose is to interact with its environment (made up of the other entities) in a fashion which provides a service to that environment. Several research communities have devoted much effort to designing methods for describing system components, for expressing properties of their collective behavior as a service, and for demonstrating correctness in a proof that a given system's behavior has stated properties. The research has been done in the context of software engineering (eg [8]), communication protocols (eg [10, 3, 20]), semantics of concurrent programs (eg [9, 18, 2]), and design of distributed algorithms (eg [4, 13, 15]); while these groups often pay little attention to one another, it is remarkable that on some key issues many different proposals are closely related. At the "Concurrency and Distribution" track of the International Workshop of Software Specification and Design (reported in [6]) supporters of several formal methods met and discussed how each method would approach an example problem.

While there was considerable disagreement between different methods on how to represent system composition, and what proof techniques to support, a significant number of research proposals have all taken the view that a single localised component (often called a "process") is best modeled as a transition system[1]; that is, there is a state space, with a transition relation indicating how the state can change, the system executes in discrete events from an initial state, taking steps each of which is in the transition relation.

In formal methods which represent components as transition systems, it is normal to define correctness of a system by giving a property that must hold in every execution of the system. There are a variety of ways to present such a property, which is the *specification* of the requirements that the environment places on the system. While some methods of formal description treat a specification as a formula in a temporal logic, and others have allowed set-theoretic descriptions of the set of allowed executions, it is noticeable that in practice even researchers who adopt these formal methods often choose a simpler way to express the specification (which is the only way in some of

[1] Of course, there is no uniformity on a preferred syntax for expressing the model; however, the various formal methods each have a syntax which has a natural semantics as a transition system

SIGSOFT '95 Washington, D.C., USA
© 1995 ACM 0-89791-716-2/95/0010...$3.50

the more pragmatic formal methods from the software engineering community). Thus among a range of formal methods, the common way to give a specification is by a *model*, that is, to present a transition system whose executions are exactly the ones desired ([6], p 20, 4th column). Of course the specification model would not be suitable as an implemented system (it is a single global component, while the real system must be distributed, fault-tolerant etc), but it provides a simple, easily-understood view that can be shared between clients of the service and those who must implement the system, as discussed in more detail in [7]. A powerful advantage of this model-based style of specification is that it allows correctness proofs to use the mapping techniques called "refinement", "simulation", or "abstraction function", where the verifier gives a correspondence between states of the implementation and those of the specification, and shows that the correspondence is preserved by each step of the system; it follows by induction that executions of the system correspond to executions of the specification model.

1.2 Specifying Liveness Properties

The bulk of work, especially by the software engineering community, in formal methods for concurrent systems has concentrated on "safety" properties, where the interest is on finite executions (often called "traces"): one shows that the system never does something wrong. It is widely recognised that "liveness" properties, which show that something right eventually occurs, are much harder to deal with. However, if one considers specifying a service in an open environment, it is clear that the applications will need to know about the extent to which the service guarantees to respond to requests, and therefore, if formal specifications are used at all they must express liveness conditions. For example, most applications would fail if they used a communication server that did not guarantee that messages were eventually delivered.

The most common way to present the liveness aspects of a specification is by temporal logic formulas [17]. This is highly expressive, that is it makes it easy to say many different conditions; indeed [17] use temporal logic to describe both safety and liveness. Even many formal methods that express safety specifications as model transition systems use temporal logic for specifying liveness [13, 5, 20]. That is, in these methods a specification involves both a model and a temporal formula, and the meaning of such a specification is a subset of the executions of the model, namely the subset that satisfy the temporal formula. However, using temporal logic for expressing liveness means that verification of liveness follows the inference-rule style rather than the mapping style that has been so easy for proving safety properties with model-based specifications. There are a few formal methods ([16, 11]) in which a model includes certain fairness conditions, by which a particular subset of the executions are naturally selected. There are several variant definitions of fairness, but the most common is that in a fair execution, any transition that is continually enabled must eventually happen. In these methods one can provide both safety and liveness in a model-based specification, by saying that a system is correct provided all its fair executions are fair executions of the specification model.

1.3 This Paper

This paper examines the usefullness of the fairness-based style for expressing liveness with model-based specifications of a service.[2] Our interest is in specifications in practice; the theoretical limits of transition fairness is discussed in [19]. We use as a testbed a service that has been previously described in the specification literature: a cancellable resource allocator [6]. We will consider a range of liveness conditions, and see to what extent it is easy to invent the model or modify one for alternative liveness constraints. In doing so we will demonstrate some of the common techniques needed to produce models with desired transition fairness executions.

We choose Input/Output Automata ("I/OA") [15, 16] as the formal method where safety is expressed by a model transition system, and liveness by the fairness properties of that model. To make comparison easier, we define an alternate formal method called Input/Output Automata with Temporal Logic ("I/OA-TL"), which expresses a transition system exactly as in I/OA, but uses temporal logic to identify the subset of all executions that will be acceptable. In Section 2, we summarise the aspects of the two formal methods; in Section 3 we introduce the service, we present a natural model as a safety specification, and then we analyse the liveness behavior of this model under the I/OA fairness definition, and discuss how to use I/OA-TL to achieve the same meaning. In Section 4 we consider alternative liveness conditions, and see how easy (or natural) it is to express them in each framework. In Section 5 we present our conclusions.

2 The Formal Methods

The *Input/Output automaton* method was defined by Lynch, Tuttle and Merritt [15] as a tool for modeling concurrent and distributed systems. Examples are given in [7] to show how the model can represent service specifications, complete systems, and components of a system. We refer the reader to the expository paper [16] for a complete development of the method, plus motivation and examples. Here, we provide a brief summary of the main aspects of the model relevant to specifications. We also define a related model which uses the same transition system for describing safety properties, but adds temporal logic to determine liveness.

2.1 Transition Systems

The core of most formal models of concurrency (and in particular, of both formal methods considered in this paper) is the transition system. This fundamental model

[2] Note that we only describe the requirements placed on the service, we are not considering how to model a complex distributed algorithm providing that service.

for a concurrent computation was first defined in [12]; for use in this paper we have added the division of the set of actions between input, output and internal ones, and the requirement that input actions are always enabled. These extra features are important in allowing a compositional semantics based only on sequences of actions.

Formally, a *transition system* T consists of four components:

1. Three sets of named actions, $in(T)$, $out(T)$ and $int(T)$, being the *input actions*, *output actions* and *internal actions* respectively. We require that these sets be pairwise disjoint. The input actions represent discrete events in the automaton that are under the control of its environment; the output actions represent events that are under the control of the automaton and may also affect the environment; and internal actions are under the control of the automaton but are not observed by the environment. We refer to the union of all three sets as $acts(T)$, the set of *actions*. Often we are only interested in those actions that relate to the environment: thus we speak of the set of *external actions*, written $ext(T)$, which is $in(T) \cup out(T)$.
2. a set $states(T)$ of *states*. Usually a state is described by assigning a value to each of a collection of typed variables.
3. a nonempty set $start(T)$ of *start states*, each being a state of the automaton. The start states represent possible initial configurations of the system.
4. a transition relation $steps(T) \subseteq (states(T) \times acts(T) \times states(T))$. An element (s', π, s) of $steps(T)$ is called a *step* of the automaton, and it represents the fact that the automaton can change from state s' to state s by performing the action π. We say that action π is *enabled* in state s' if there is a transition (s', π, s) in $steps(T)$. In order to reflect the fact that input actions are under the control of the environment, we require that for every state s' and input action π, π is enabled in s'.

An *execution* of T is a finite sequence $s_0 \pi_1 s_1 \pi_2 \ldots \pi_n s_n$ or an infinite sequence $s_0 \pi_1 s_1 \pi_2 \ldots \pi_n s_n \ldots$ of alternating states and actions of A such that s_0 is a start state, and $(s_i, \pi_{i+1}, s_{i+1})$ is a step of T for every i. A state is said to be *reachable* in T if it is the final state of a finite execution of T.

The *behavior* of an execution α of T is the subsequence of α consisting of external actions, and is denoted by $beh(\alpha)$. That is, to form the behavior, we ignore all states and internal actions in the execution. We say that β is a *behavior* of T if β is the behavior of an execution of T.

An important operation on behaviors or other sequences is *projection*. If α is a sequence (of elements of any alphabet) and Φ is a set of elements, we write $\alpha|\Phi$ for the subsequence of α consisting of the occurrences of those elements in the set Φ. Thus if α is an execution of transition system T, then $beh(\alpha) = \alpha|ext(T)$.

2.2 Input/Output Automata

An *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of a transition system $trans(A)$, together with an equivalence relation $part(A)$ on $out(trans(A)) \cup int(trans(A))$, having at most countably many equivalence classes. Each equivalence class represents a group of actions under the control of the automaton, and fairness will be required between separate classes when the automaton executes.

An *execution* of A is just an execution of the transition system. A *fair execution* of an automaton A is defined to be an execution α of the transition system for which the following condition holds for each class C of $part(A)$: if α is finite, then no action of C is enabled in the final state of α, and if α is infinite, then either α contains infinitely many events from C, or else α contains infinitely many occurrences of states in which no action of C is enabled. Thus, a fair execution gives "fair turns" to each class of $part(trans(A))$. Informally, one class of $part(A)$ typically consists of all the actions that are controlled by a single subsystem within the system modeled by automaton A, and so fairness means giving each such subsystem regular opportunities to take a step under its control, if any is enabled. In the common case that there is no lower level of structure to the system modeled by A (when all locally-controlled actions are in a single class of $part(A)$), a fair execution is an execution in which infinitely often the automaton is given an opportunity to take a locally controlled action if any is enabled. That is, the fair executions are those executions where an infinite number of locally-controlled actions happen, those where infinitely often the automaton is in a state with no enabled locally-controlled action, and those which are finite and have final state in which no locally-controlled action is enabled. In the other common case, each locally-controlled action is placed in a separate class, and then a fair execution is an execution where any locally-controlled action that is continually enabled after some point must happen infinitely often.

We say that β is a *behavior* of A if β is the behavior of an execution of A. We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A and we denote the set of fair behaviors of A by $fairbehs(A)$. When an algorithm is modeled as an I/O automaton, it is the set of fair behaviors of the automaton that reflect the activity of the algorithm that is important to users.

The formal method also includes a definition for system composition, but it is not needed in this paper.

2.2.1 Correctness

Fundamental to any use of a formal method in reasoning is a definition of the relationship between a service specification and a system that provides the service. The formal method allows specifications to be given by presenting sets of allowed fair behaviors, but more often, in the model-based style, both specification and system are represented as automata. We say that system A *solves* specification B provided that $in(trans(A)) = in(trans(B))$,

$out(trans(A)) = out(trans(B))$, and $fairbehs(A) \subseteq fairbehs(B)$. That is, the system has the external interface expected in the specification, and every fair behavior of the system is among the fair behaviors of the specification. Notice that we allow the situation where the specification has fair behaviors that are *not* exhibited by the system. Also, notice that internal actions and state spaces may be quite different, between specification and system.

2.3 I/OA with Temporal Logic

In order to compare the expressive power of fairness-based service specifications against temporal logic, we introduce a formal method[3] called *Input-Output Automata with Temporal Logic*. In this method, a specification A consists of a transition system $trans(A)$ (presented by sets of input actions, output actions, internal actions, states, start states, and steps) and also a temporal formula $temp(A)$. In other words, an I/OA-TL specification is just like an automaton except that the equivalence relation is replaced by a formula. The temporal formula is an expression in a temporal logic where atomic formulas[4] are the action names, and propositional connectives are used[5], as well as several temporal operators defined in [17]. These include three *future* operators – two unary temporal operators $\Box$ and $\Diamond$ and one binary (infix) temporal operator $\mathcal{U}$. Informally $\Box$ means "always", $\Diamond$ means "eventually", and $\mathcal{U}$ means "until". We also use three *past* operators – two unary temporal operators $\boxminus$ and $\diamondminus$ and one binary (infix) temporal operator $\mathcal{S}$. Informally $\boxminus$ means "has always been", $\diamondminus$ means "once", and $\mathcal{S}$ means "since".

A sequence of actions is a behavior of an I/OA-TL specification exactly when it is a behavior of the transition system. A *live behavior* of an I/OA-TL specification is a sequence that meets two conditions: it is a behavior of the transition system, and also the first place in the sequence is a model for the temporal formula; we write the set of live behaviors of A as $livebehs(A)$.

To be completely precise, we follow the usual temporal logic definitions [17] to describe when a place j in a sequence $\beta = \pi_1\pi_2\ldots$ is a model for a formula Φ (written $(\beta, j) \models \Phi$); this is done by structural induction. Since we are mostly concerned with the first place in the sequence, we write $\beta \models \Phi$ as an abbreviation for $(\beta, 1) \models \Phi$.

- if Φ is an atomic formula (that is, the name of an action π) then $\beta \models \Phi$ iff π is the j-th action in β (that is, $\pi_j = \pi$)
- if $\Phi = \Phi_1 \vee \Phi_2$ then $(\beta, j) \models \Phi$ iff either $(\beta, j) \models \Phi_1$ or $(\beta, j) \models \Phi_2$ (and similarly for other propositional connectives)
- if $\Phi = \Box\Phi_1$ then $(\beta, j) \models \Phi$ iff every later[6] place in the sequence is a model for Φ_1 (that is, when for every k greater than or equal to j, $(\beta, k) \models \Phi_1$)
- if $\Phi = \Diamond\Phi_1$ then $(\beta, j) \models \Phi$ iff some later place of β is a model for Φ_1 (that is, when for some k greater than or equal to j, $(\beta, k) \models \Phi_1$)
- if $\Phi = \Phi_1 \mathcal{U} \Phi_2$ then $(\beta, j) \models \Phi$ iff there is some later place for which Φ_2 holds, and Φ_1 holds for every place from the current one up to (but not including) that one. More mathematically, this is when there exists k greater than or equal to j such that $(\beta, k) \models \Phi_2$ and also for every i such that $j \leq i < k$, $(\beta, i) \models \Phi_1$
- if $\Phi = \boxminus \Phi_1$ then $(\beta, j) \models \Phi$ iff every earlier place in the sequence is a model for Φ_1 (that is, when for every k less than or equal to j, $(\beta, k) \models \Phi_1$)
- if $\Phi = \diamondminus\Phi_1$ then $(\beta, j) \models \Phi$ iff some earlier place of β is a model for Φ_1 (that is, when for some k less than or equal to j, $(\beta, k) \models \Phi_1$)
- if $\Phi = \Phi_1 \mathcal{S} \Phi_2$ then $(\beta, j) \models \Phi$ iff there is some earlier place for which Φ_2 holds, and Φ_1 holds for every place from (but not including) that one up to the current one. More mathematically, this is when there exists k less than or equal to j such that $(\beta, k) \models \Phi_2$ and also for every i such that $k < i \leq j$, $(\beta, i) \models \Phi_1$

To clarify this notation, let us present some simple examples. The formula $\Box(\pi \vee \psi)$ applies to β when every action in the sequence is either π or ψ; the formula $\Diamond\pi$ says that π occurs somewhere in the sequence; the formula $\Box\Diamond\pi$ says that π occurs infinitely often in the sequence; the formula $\Box(\pi \Rightarrow (\Diamond\psi))$ says that every π is followed eventually by ψ; and $(\neg\pi)\mathcal{U}\psi$ says that ψ occurs in the sequence, and that its first occurrence comes before any occurrence of π.

Even when I/OA-TL is used to capture a specification, an implemented system will be modelled by an automaton (with fairness used to determine allowed executions). In this situation, we will say the automaton A *solves* an I/OA-TL specification B when $in(trans(A)) = in(trans(B))$, $out(trans(A)) = out(trans(B))$, and $fairbehs(A) \subseteq livebehs(B)$.

3 The Cancellable Resource Allocator

The 7th International Workshop on Software Specification and Design (IWSSD-7) was held in Redondo Beach California in December 1993. As part of the track on Concurrency and Distribution, participants familiar with a range of formal specification methods ([9, 15, 4]) were

[3] This method is presented only to provide a level-playing field for the comparisons in this paper, it is not a serious alternative methodology

[4] In the other temporal logic formal methods known to the author, both safety and liveness are expressed in terms of the sequence of states through which the model passes (with this state space thus accessible to all the environment). This paper uses sequences of events, as is done in process algebra methods, in the belief that this is more appropriate to model distributed systems where each service is implemented by (one or more) distinct processes, each with its own private address space.

[5] For convenience, when action names include parameters, we use existential and universal quantification over the values of one or more parameters as an abbreviation for a large disjunction or conjunction of formulas, one for each value of the parameter

[6] Throughout these descriptions, the words "later' and "earlier" are not *strict*; that is, they include the place itself

asked to specify a system using their preferred technique. The system is described as follows, quoted from [6].

"A non-shareable resource R is used by a set of user processes $U_1,\ldots,U_n$. Access to R is managed by an allocator process A_R. Before using R, user process U_i request access from A_R, which responds with an allocation indication when R becomes free. When a user process U_i is finished with R, it signals the allocator that it has released the resource. At any point between requesting the resource and becoming aware that it has been allocated the resource, a user process may cancel its request. A solution should not preclude U_i and A_R from being physically distributed. The system must ensure that R is never allocated to more than one U_i at a time, and should minimise the time that R is allocated to a U_i that has released it or cancelled the request for it."

This problem is similar to the resource allocation problems whose specifications in temporal logic are discussed by Manna and Pnueli [17], but the possibility of cancelling a request is not present in their work; this makes our I/OA-TL descriptions more complicated than those in [17].

In the I/O automaton method, and thus also in using I/OA-TL, the fundamental step in specification is to identify and name the actions by which a component interacts with its environment. We must also determine whether each is controlled by the component (an output of the component) or controlled by the environment (an input of the component). In the case of the resource allocator specification, one can choose the following actions:
Request(i): client U_i requests access (input of the allocator)
Cancel(i): client U_i cancels outstanding request (input of the allocator)
Grant(i): client U_i is informed that it has access (output of the allocator)
Finish(i): client U_i releases resource (input of the allocator).

Note that **Request**(1) is a different action from **Request**(2) and so on; formally there is no significance in the fact that they have similar names (although in most cases the transition relation will follow a uniform pattern that is most easily described using a free variable i in **Request**(i).

Here is the transition system for a natural model-based specification of the safety requirements on the resource allocator. We will refer to the specification as T_{safe}. The state space is given by all possible assignments (that respect type declarations) to the following variables: *req* an array of booleans initially false, and *grant* an array of booleans initially false. The meaning of $req[i] = true$ is that U_i has made a request that has not yet been satisfied or cancelled; the meaning of $grant[i] = true$ is that U_i is currently allowed to access R.

Next we give the transition relation, in Figure 1. Syntactically, each action type is described here with a precondition and an effect. Formally, the transition relation is actually a set of triples (s', π, s): the syntax describes the set of all such triples where π is the named action, s' is some state in which the precondition is true, and s is a state which is produced from s' by performing the assignments listed as the effect. When no precondition is listed, the default is precondition=*true*, as must happen for each input action (since these are always enabled). As it happens, this automaton has no internal actions, but that is not a necessary feature of a model-based specification.

It is easy to see that this is a direct expression of the English description of the service. For example, the input actions **Request**(i) and **Cancel**(i) simply alter the i-th entry of *req* to reflect that a request is now outstanding or not. The input **Finish**(i) adjusts the i-th entry of *grant* to reflect that the resource is no longer allocated. These inputs are under the control of the users, not that of the allocator, so they are always enabled. The output action **Grant**(i) can occur whenever there is an outstanding request from U_i (expressed in the first clause in the precondition) and also no user is currently allocated the resource (expressed in the second clause of the precondition). Its effect is simply to record that the resource has been allocated and that the request is no longer outstanding.

3.1 Fairness and the Transition System

Now we consider what happens if we use the transition system T_{safe} as the basis of an I/O automaton; that is, we explore which behaviors are fair.

Suppose, as is common, that each output action is in a separate class of the equivalence relation. That is, let A_1 denote the automaton with $trans(A_1) = T_{safe}$ and $part(A_1)$ is the diagonal relation on the output actions. We want to understand the liveness requirement on the service described by the fair executions of this model. It turns out that this model cannot deadlock: that is, if there is any user who makes a request and does not cancel it, then the allocator must eventually grant the resource to someone.

As an example, the following infinite execution of A_1 (in which users 1 and 3 repeatedly request and then cancel) is not fair, since user 2 never cancels its request but no grant occurs.
Request(1)
Request(2)
Request(3)
Cancel(1)
Request(1)
Cancel(3)
Request(3)
Cancel(1)
Request(1)
Cancel(3)
Request(3)
....

However, the model does allow any individual user to starve. There are fair executions in which a particular user makes a request which is never granted or cancelled. While fairness forces some user to be granted access, there is no certainty that the particular request will ever be

Request(i)
Effect:
$req[i] \leftarrow true$

Cancel(i)
Effect:
$req[i] \leftarrow false$

Finish(i)
Effect:
$grant[i] \leftarrow false$

Grant(i)
Precondition:
$req[i]$
$\nexists j : grant[j]$
Effect:
$grant[i] \leftarrow true$
$req[i] \leftarrow false$

Figure 1: Transition System Safety Specification: T_{safe}

granted, as fairness ensures that an action from the equivalence class will occur *provided the class is continuously enabled.* Whenever one user gains access, all other grants become disabled, so fairness is not violated even when one particular grant action never occurs.

One fair execution in which user 2 is starved is the following, which we call β_{starve}:
Request(2)
Request(3)
Grant(3)
Finish(3)
Request(1)
Grant(1)
Finish(1)
Request(3)
Grant(3)
Finish(3)
Request(1)
Grant(1)
Finish(1)
....

The reason β_{starve} is fair is that every time the resource is allocated, all other **Grant** actions become disabled. Thus **Grant**(2) is not continuously enabled (even though U_2 is continuously waiting), so even with **Grant**(2) being in a separate class of the equivalence relation, fairness does not ensure that the action occurs.

Thus the most natural I/OA model is convenient for the service provider (it is easy to find algorithms if they can starve users), but it is not very pleasant for the client.

For comparison, we also give an I/OA-TL specification of the deadlock-free, but starvation-prone service by using the same natural transition relation T_{safe}, and adding an appropriate liveness formula. To keep things readable, let us define some abbreviations. We use $held(i)$ to denote $(\neg\textbf{Finish}(i))\mathcal{S}\textbf{Grant}(i)$; this formula holds at any place where user i is allocated the resource. We use $asking(i)$ to denote $(\neg(\textbf{Cancel}(i) \vee \textbf{Grant}(i)))\mathcal{S}\textbf{Request}(i)$; this formula holds at any place where user i is requesting the resource. That is, these formulas express the same facts as $grant[i] = true$ and $req[i] = true$, respectively, but they are written using temporal logic on sequences of actions instead of state components. With this notation, the presence of deadlock at a place can be represented by the formula $(\exists i\, \Box asking(i)) \wedge (\Box(\neg(\exists j\, held(j))))$; in English one would say that deadlock is when some user requests forever and it is always the case that no user has the resource. Thus the condition that deadlock never happens is $\Box\neg((\exists i\, \Box asking(i)) \wedge (\Box(\neg(\exists j\, held(j)))))$, which is easily transformed to the equivalent formula $\Box((\exists i\, \Box asking(i)) \Rightarrow (\Diamond(\exists j\, held(j))))$.

We can easily prove that the given I/OA-TL specification has the set of its live behaviors being exactly the fair behaviors of A_1. To do so, note from the transition relation that the output action **Grant**(i) is enabled exactly at the places where $asking(i) \wedge (\forall j\ \neg held(j))$ holds. We will abbreviate this to $enabled(i)$. In general, an execution is not fair when some output is eventually enabled forever without occuring; for A_1, each output is disabled immediately after it occurs. Thus for A_1, an execution is not fair exactly if some output is eventually always enabled. Thus the fair executions are described by $\forall i\, \Diamond\Box enabled(i)$. Simple temporal equivalences given in [17] show this is equivalent to the formula given above for no deadlock.

3.2 Other Automata for the Same Transition System

One might think to change the specification by taking the same transition system with a different equivalence relation. For example one could place all the output actions in the same equivalence class.

No matter how the equivalence relation is defined, the specification allows starvation. The reason is that whenever one user is granted the resource, all the output actions are disabled. Thus no matter what equivalence relation is chosen, any execution with infinitely many **Grant** actions has infinitely many states in which no output is enabled; that is, the execution is fair.

However it is *not* the case that changing the equivalence class has no effect on the set of fair behaviors. When all output actions are in a single equivalence class, the following is not a fair behavior (even though it is fair for A_1,

as it does not involve a deadlock).
Request(1)
Request(3)
Cancel(1)
Request(1)
Cancel(3)
Request(3)
Cancel(1)
Request(1)
Cancel(3)
Request(3)
.... In this execution, each request is eventually cancelled, but the interleaving is such that at each instant, one user has a request that is outstanding (the user involved is different at different instants). Thus at every state, there is some action that is enabled and is in the single equivalence class; however no output ever occurs.

We can give an equivalent in I/OA-TL to the specification with all output actions in a single class. We use the same transition system, and as temporal formula we take $\Box\Diamond((\exists i\ \mathbf{Grant}(i)) \vee \neg(\exists i\ \mathit{enabled}(i)))$. This expresses directly the definition of fair executions. We remark that one *cannot* use a similar agrument to claim that any automaton has an equivalent in I/OA-TL, since in general one can't always find a temporal formula that captures when an action is enabled.

4 Variant Specifications

In this section we will consider in turn several specifications which make stronger or weaker guarantees about when users' requests are granted. For each, we will show how to give an automaton, and also a specification in I/OA-TL.

4.1 Deadlock Allowed

The first variant we consider is when we wish to be completely permissive; that is we want to find accept every (finite or infinite) behavior of T_{safe}, even those in which deadlock occurs. In I/OA-TL this is trivial: we simply take the transition system T_{safe} and the temporal formula *true*.

Transition system fairness means that continuously enabled output actions eventually occur. Therefore, to make a automaton whose fair behaviors include many sequences in which outputs do not occur, we must change the underlying transition system so that each output is periodically disabled. This is easy to arrange. We define a new transition system $T - \mathit{pause}$. The actions of T_{pause} are all the actions of T_{safe} together with two internal actions **stop** and **go**. The state variables of T_{pause} are the state variables of T_{safe} together with *status* which is a boolean, initailly true.

The transition relation is given in Figure 2. The only changes from that of T_{safe} are the two internal actions and also an extra precondition, so *status* must be true whenever a **Grant** output occurs.

Finally we take the transition system T_{pause} with the equivalence relation in which each output and internal action is in a separate equivalence class. It is easy to see that any behavior of this automaton is a behavior of T_{safe}, just by taking the execution in which it arises and ignoring the *status* component in the state, and the internal steps. To show that any behavior of T_{safe} is actually a fair behavior of T_{pause}, we can take an execution in which it arises, and replace each state s of T_{safe} in that execution by a brief sequence, which begins with a state of T_{pause} which has $\mathit{status} = \mathit{true}$ and has other components identical to s; followed by the internal action **stop** followed by a state with $\mathit{status} = \mathit{false}$ and otherwise identical to s; followed by **go** followed by the state which has $\mathit{status} = \mathit{true}$ and has other components identical to s. If the original sequence was finite, ending in state s, we extend the result of the replacement to an infinite sequence by adding an endless pattern consisting of repetitions of the following: the internal action **stop** followed by a state with $\mathit{status} = \mathit{false}$ and otherwise identical to s; followed by **go** followed by the state which has $\mathit{status} = \mathit{true}$ and has other components identical to s. The result of this transformation is clearly an execution of T_{pause}; furthermore the result is fair, since every output action is infinitely often disabled (immediately after **stop**) and each internal action is performed infinitely often.

4.2 No Starvation

Next we consider restricting the allowed behaviors, rather than increasing the set as in the previous subsection. Suppose we wish to write a specification that says "the system gives mutual exclusion, and no user is allowed to starve". That is, one might want to specify that if a user makes a request and does not cancel it, then eventually that user must be allocated the resource. To do this in I/OA-TL, we start with the original transition system T_{safe}, and add an appropriate temporal formula. However, one must be careful. To say "user i does not starve" is given as $\Box(\mathbf{Request}(i) \Rightarrow \Diamond(\mathbf{Cancel}(i) \vee \mathbf{Grant}(i)))$, that is, any request is eventually followed by a cancel or grant for user i. However, it is not correct to simply add this as a liveness condition to our simple transition system. One must be wary: the guarantee can't be met unless there are also liveness requirements placed on the users, that each eventually releases the resource after being granted it. This aspect (of conditional specification) is considered in the TLA formal method by Abadi and Lamport [1]. Thus in I/OA-TL, the liveness condition must be $\Box((\forall i(\mathbf{Grant}(i) \Rightarrow \Diamond\mathbf{Finish}(i)))) \Rightarrow \Box(\forall i(\mathbf{Request}(i) \Rightarrow \Diamond(\mathbf{Cancel}(i) \vee \mathbf{Grant}(i))))$.

To present a fairness-based specification for this idea, that no user may be starved, is not at first obvious. Since fairness makes each action happen when it is continuously enabled, our idea is to make each grant action eventually always enabled; however, each grant is disabled whenever another user holds the resource, so we think we should prevent other users from getting the resource until the given user has done so. This is done by including in the state variables which determine how often each request

Request(i)
Effect:
$req[i] \leftarrow true$

Cancel(i)
Effect:
$req[i] \leftarrow false$

Finish(i)
Effect:
$grant[i] \leftarrow false$

stop
Precondition:
$status$
Effect:
$status \leftarrow false$

Grant(i)
Precondition:
$req[i]$
$\nexists j : grant[j]$
$\neg status$
Effect:
$grant[i] \leftarrow true$
$req[i] \leftarrow false$

go
Precondition:
$\neg status$
Effect:
$status \leftarrow true$

Figure 2: Transition System Allowing Deadlock: T_{pause}

can be overtaken. We extend the state by variables *pass*, which a two-dimensional array of integers, initially all zero, and *max*, a two-dimensional array of non-negative integers, initailly all zero. The meaning of $pass[i,j] = n$ is that U_i is waiting for the resource, and that U_j has been allocated the resource n times since U_i made its request. The meaning of $max[i,j] = n$ is that during U_i's current wait, there are only allowed to be n grants to U_j.

The transition relation is given in Figure 3. The changes from the original model T_{safe} are that each time a user process starts to wait, an arbitrary value is chosen for the limits on how often it can be overtaken. Note that the changes are that $pass[i,j]$ is reset to zero each time U_i ceases to be waiting (by being allocated the resource, or by cancelling), that it is increased each time U_j is granted the resource while U_i is waiting, and that a precondition prevents the resource being allocated to a user if any other user has already been overtaken by it too often.

Notice that the model is *nondeterministic*, not just in having more than one action enabled in a state (as we have seen before), but also in having more than one state possible as the new state after a given action occurs. This is expressed in the syntax by the free variable x in the effect of **Request**; the semantics is that the transition relation includes all (s', π, s) where there exists a value for the free variable x which satisfies the "where" clause and for which applying the effect assignments changes s' into s.

4.3 One-bounded Overtaking

The most obvious way to make the specification useful to clients is to demand that requests are handled in FIFO order: if U_i makes a request, then it will be granted or cancelled before any grant is made to a user which requested after U_i. This is trivial to state in a model-based specification, by keeping a queue of requesting users, and always granting only to the head of the queue. Unfortunately, this specification is not implementable in a distributed system, where there is no way to detect the true order of events in different places. Thus we do not present this here.

The normal way to capture an idea close to FIFO in a distributed system is to require that each user who does not cancel his request must be granted the resource before he has been overtaken more than once by each other user. Thus an allowed fair behavior would be:
Request(1)
Request(2)
Grant(2)
Finish(2)
Request(3)
Grant(3)
Finish(3)
Grant(1)
Finish(1).

However, the following would not be allowed, since user 2 is granted the resource twice while user 1 waits:
Request(1)
Request(2)
Grant(2)
Finish(2)
Request(2)
Grant(2)
Finish(2)
Grant(1)
Finish(1).

Notice that this variant involves conditions that can be detected in finite behaviors (not just infinite ones); in technical terms it is not a pure liveness property conjoined with the "mutual exclusion" safety property. It is easy to

Request(i)
Effect:
  $req[i] \leftarrow true$
  doforall j: $max[i,j] \leftarrow x$
    where $x \geq 0$

Cancel(i)
Effect:
  $req[i] \leftarrow false$
  doforall j: $pass[i,j] \leftarrow 0$

Finish(i)
Effect:
  $grant[i] \leftarrow false$

Grant(i)
Precondition:
  $req[i]$
  $\nexists j : grant[j]$
  $\nexists j : req[j] \wedge pass[j,i] \geq max[j,i]$
Effect:
  $grant[i] \leftarrow true$
  $req[i] \leftarrow false$
  doforall j: if $req[j]$ then
    $pass[j,i] \leftarrow pass[j,i] + 1$
  doforall j: $pass[i,j] \leftarrow 0$

Figure 3: Transition System With No Starvation: T_{bound}

achieve this effect by taking the transition relation T_{bound} above, and always choosing 1 as the value of $max[i,j]$.

Expressing 1-bounded waiting is quite convoluted in temporal logic, where counting is not easy. Based on Manna and Pnueli[17] we give the following formula, which we present in stages. First define $p\mathcal{W}q$ to mean $(p\mathcal{U}q) \vee \Box p$. Next define $atmostonce(i,j)$ by $\mathbf{Request}(i) \Rightarrow (\neg\mathbf{Grant}(j)\mathcal{W}(\mathbf{Grant}(j)\mathcal{W}(\neg\mathbf{Grant}(j)\mathcal{W}(\mathbf{Grant}(i) \vee \mathbf{Cancel}(i)))))$; this is a predicate which is true at any place where i requests the resource and is not subsequently overtaken twice by j while waiting, and also at places where i does not request the resource. We can now express 1-bounded waiting as $\Box\forall i,j\ atmostonce(i,j)$, which we abbreviate to *onebounded*. Thus to get the liveness formula to combine with T_{safe} one would say $(\Box\forall i\,(\mathbf{Grant}(i) \Rightarrow \Diamond\mathbf{Finish}(i))) \Rightarrow (onebounded \wedge (\Box\forall i\,(\mathbf{Request}(i) \Rightarrow (\mathbf{Grant}(i) \vee \mathbf{Finish}(i)))))$.

5 Conclusions

This paper has shown that it is feasible to use model-based service specifications (appropriate for safety conditions), and include liveness as the consequence of fair execution of the model. This is in contrast to the more common technique, where a model-based safety specification is augmented with temporal logic formulas to express liveness. More experience is needed to determine whether the single model style is more convenient for specifiers and verifiers of the typical services needed in building reactive systems, compared with the traditional mixed approach where proofs use mapping techniques for safety and inference rules for liveness.

By examining a case study of a cancellable resource allocator, we have seen how a range of different liveness conditions can be expressed using a model-based specification in a formal method where executions have fairness characteristics. The most natural model implied a rather unsatisfactory liveness condition, where global deadlock was prevented but starvation of some individual users was allowed. We have shown how to vary the model to obtain more or less infinite fair behaviors. To make all infinite behaviors fair, we repeatedly disabled all activity then reenabled it. To express the absence of starvation as a specification, we kept track of how often each request was overtaken, and non-deterministically chose limits on these quantities. We could also express 1-bounded overtaking, which is an approximation to FIFO ordering and yet is achievable in a distributed system. We have also shown how to express these same conditions in temproal logic.

It is possible that in many practical cases, liveness may be subsumed by more precise timing guarantees. These can also be expressed in a model based style, as shown in [14].

References

[1] Abadi, M. and Lamport, L., "Open Systems in TLA" *Proc. ACM Symposium on Principles of Distributed Computing*, pp 81–90, August 1994.

[2] Baeten, J. and Weijland, W., *Process Algebra*, Cambridge University Press, 1990.

[3] Bolognesi, T. and Brinksma, E., "Introduction to the ISO specification language Lotos", *Computer Networks and ISDN Systems*, vol. 14, pp. 25–59, 1987.

[4] Chandy, M. and Misra, J., *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[5] Duke, R. and Smith, G., "Temporal Logic and Z Specifications" *Australian Computer Journal*, 21(2):62-66, May 1989.

[6] Feather, M. and van Lamsweerde, A., "Succeedings of the 7th International Workshop on Software Specification and Design" *Software Engineering Notes*, 19(3):18-22, July 1994.

[7] Fekete, A., "Formal Models of Communication Services: A Case Study" *IEEE Computer*, 26(8):37-47, August 1993.

[8] Hayes, I., *Specification Case Studies*, Prentice Hall International, 1987.

[9] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall International, 1985.

[10] Holzmann, G., *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1991.

[11] Jonsson, B., "Compositional Specification and Verification of Distributed Systems" *ACM Transactions on Programming Languages and Systems*, 16(2):259-303, 1994.

[12] Keller, R., "Formal Verification of Concurrent Programs" *Communications of the ACM*, 19(7):371-384, 1976.

[13] Lamport, L., "The Temporal Logic of Actions" *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, 1994.

[14] Lynch, N., and Attiya, H., "Using Mappings to Prove Timing Properties" *Distributed Computing*, 6(2):121-139, 1992.

[15] Lynch, N., and Tuttle, M., "Hierarchical Correctness Proofs for Distributed Algorithms" *Proc. ACM Symposium on Principles of Distributed Computing*, pp 137-151, August 1987.

[16] Lynch, N., and Tuttle, M., "An Introduction to Input/Output Automata" *CWI Quaterly*, 2(3):219-246, September 1989.

[17] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems* Springer-Verlag, 1992.

[18] Milner, R., *Communication and Concurrency*, Prentice Hall International, 1989.

[19] Reingold, N., Wang, D.-W., and Zuck, L., "Games I/O Automata Play" in *Proc. Third International Conference on Concurrency Theory*, August 1992, Springer-Verlag LNCS 630.

[20] Shankar, A. and Lam, S., "A stepwise refinement heuristic for protocol construction", *ACM Transactions on Programming Languages and Systems*, vol. 14, pp. 417–461, July 1992.