Composing Fair Objects

G.W. Hamilton
School of Computer Applications
Dublin City University
Ireland
hamilton@compapp.dcu.ie

J.P. Gibson
Department of Computer Science
NUI Maynooth
Ireland
pgibson@cs.may.ie

D. Méry
Université Henri Poincaré
Nancy
France
mery@loria.fr

Abstract

When specifying large systems, we would like to be able to specify small components independently, and to be able to compose them in such a way that their specified properties are preserved. We have previously proposed the concept of a fair object, which incorporates the specification of both safety and liveness properties, as a suitable such unit of specification. Unfortunately, however, liveness properties of fair objects are often not preserved under composition. In this paper, we define a simple test to determine whether the liveness properties of fair objects are preserved under composition. We then show how liveness properties can be restored in some cases when they are broken, through the addition of fairness constraints.

1 Introduction

The temporal logic of actions (TLA) [10] is a formalism for specifying and reasoning about concurrent systems. TLA can be used to specify both *safety* properties (nothing bad will happen) and *liveness* properties (something good will happen). Although crude structural mechanisms have been proposed for TLA [12], it does not provide the high-level composition mechanisms which are essential for synthesising and analysing complex behaviour. These mechanisms are provided by the object-oriented paradigm, but most object-oriented formalisms are based on the specification of safety properties and do not incorporate liveness properties.

In [7], we proposed the concept of *fair objects*, which combine temporal semantics and object-oriented concepts in a complementary fashion. This concept is not new; similar constructs have been proposed earlier [5, 3, 1, 2]. In this paper, we study the effect of object composition on the liveness requirements of the objects being composed.

When following an object-oriented approach to specifying the behaviour of concurrent systems, we would like to be able to specify the liveness requirements of each object locally. However, an object must be viewed as an open system which relies on its environment to ensure that its local requirements are satisfied. For this reason, open systems are often specified using an assumption/guarantee (also called rely/guarantee and assumption/commitment) style of specification [9, 14, 15, 1, 16] which asserts that an object guarantees to meet its requirements under the assumption that its environment also meets its requirements.

When objects specified using the assumption/guarantee style are composed, the resulting composition must be checked to ensure that the assumptions about the environment specified for each object are still satisfied. This normally involves discharging some reasonably complex proof obligations. If the environment assumptions for each composed object are preserved, we say that these objects are *polite* [3, 7]. As pointed out in [1], this approach works best when the environment assumptions are safety properties, as liveness requirements are often broken under composition.

The approach which we take in this paper is to accept that liveness requirements are likely to be broken under composition (we define a simple mechanism to determine whether this is the case), and to show how these liveness requirements can be restored through the addition of fairness constraints; this is the main contribution of this paper.

The remainder of this paper is structured as follows. In

Section 2, we give a quick overview of TLA, and in Section 3 we show how to extend TLA with the concept of fair objects. In Section 4, we define a number of different levels of liveness for fair objects, and in Section 5 we describe the fairness constraints which can be placed on them. In Section 6, we show how fair objects can be composed, and in Section 7 we define the different levels of politeness which objects can exhibit under composition. Section 8 concludes.

2 A Quick Overview of TLA

Temporal logic extends classical logic by handling modalities such as fairness and eventuality. TLA [10] is a temporal logic of actions for specifying and reasoning about concurrent systems. TLA has four levels:

1. Constants: this level is concerned with formulas which are state-independent (*constant*). The variables which are used in this level are called *rigid variables* and cannot change values between states.

2. States: this level allows reasoning about individual states. The formulas in this level can either be *state functions* (non-boolean functions) or *state predicates* (boolean expressions). The variables used in this level can be *flexible variables*, which can change values between states.

3. Pairs of States: this level concerns reasoning about pairs of states. The formulas in this level can either be *transition functions* or *transition predicates* (*actions*). The variables used in this level can be primed. Unprimed variables refer to the old state, primed variables to the new. For a state function or predicate F, F' is obtained by replacing each flexible variable v in F by v'.

   An action $\mathcal{A}$ is a relation between states which assigns a boolean to $s[\![\mathcal{A}]\!]t$ for states s and t, where s is the old state and t is the new state. This is called an "$\mathcal{A}$ step" if it is true. $s[\![enabled\ \mathcal{A}]\!]$ is true for state s if it is possible to take an $\mathcal{A}$ step starting in that state.

   Stuttering steps are steps in which specified variables do not change. These are used to help show equivalences between behaviours. For an action $\mathcal{A}$ and a state function f, a stuttering step is written as follows:

$$[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f' = f)$$

   A step corresponding to an action $\mathcal{A}$ in which the variables in a state function f do change is written as follows:

$$< \mathcal{A} >_f \triangleq \mathcal{A} \wedge (f' \neq f)$$

4. Sequences of States: the fourth level allows reasoning about behaviours, which are infinite sequences of states. A behaviour is denoted by $< s_0, s_1, s_2, ... >$, where s_0 is the first state, s_1 is the second, etc.

   An action $\mathcal{A}$ is true iff the first pair of states in the behaviour is an $\mathcal{A}$ step.

$$< s_0, s_1, s_2, ... > [\![\mathcal{A}]\!] \triangleq s_0[\![\mathcal{A}]\!]s_1$$

The specification of a system in TLA has the following form:

$$\exists x : Initial \wedge \Box[N]_x \wedge F$$

where $Initial$ is a state predicate describing the initial state, N is a transition predicate describing the possible steps, x is a state function indicating the variables which cannot change in any step other than a N step, and F is the conjunction of fairness conditions.

3 Fair Objects

A crude module structuring mechanism has been proposed for TLA [12], but in [7] we proposed the concept of fair objects which provide higher level composition mechanisms. Similar to the way in which a system is specified in TLA, the specification of a fair object has the following form:

$$\exists x : Initial \wedge \Box[Ext \vee Int]_x \wedge F$$

where x describes the variables encapsulated within the object, $Initial$ is a state predicate describing the initial state of the object, Ext is a transition predicate describing the possible external actions of the object, Int is a transition predicate describing the internal actions of the object, and F gives the fairness constraints on the internal actions of the object (we cannot place fairness constraints on the external actions of an object, as we cannot enforce how often these should be taken).

As an example of a fair object, consider a double-ended queue as shown in Figure 1.

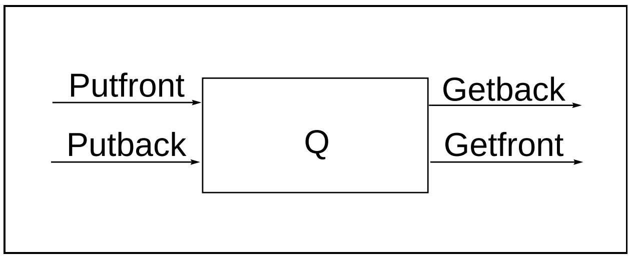

Figure 1. Double-Ended Queue

The queue object provides operations to put or get elements to the front or back of the queue (these are all external actions). This could be defined as follows in TLA:

$$
\begin{aligned}
Addfront(n) \triangleq\ & \wedge\ Len(contents) < 100 \\
& \wedge\ contents' = < n > \circ\ contents
\end{aligned}
$$

$$
\begin{aligned}
Addback(n) \triangleq\ & \wedge\ Len(contents) < 100 \\
& \wedge\ contents' = contents \circ < n >
\end{aligned}
$$

$$
\begin{aligned}
Getfront(n) \triangleq\ & \wedge\ Len(contents) > 0 \\
& \wedge\ contents = < n > \circ\ contents'
\end{aligned}
$$

$$
\begin{aligned}
Getback(n) \triangleq\ & \wedge\ Len(contents) > 0 \\
& \wedge\ contents = contents' \circ < n >
\end{aligned}
$$

$$
\begin{aligned}
Step \triangleq \exists n \in Nat :\ & \vee\ Addfront(n) \\
& \vee\ Addback(n) \\
& \vee\ Getfront(n) \\
& \vee\ Getback(n)
\end{aligned}
$$

$$Init \triangleq contents = <>$$

$$Spec \triangleq \exists contents \in Seq(Nat) : Init \wedge \Box[Step]_{contents}$$

The state variable $contents$ is a sequence which holds the elements of the queue (up to a maximum of 100 elements). We would like to guarantee that the actions $Getfront$ and $Getback$ will always eventually be enabled, but this relies on at least one of the external actions $Putfront$ and $Putback$ always eventually taking place. Similarly, we would like to guarantee that the actions $Putfront$ and $Putback$ will always eventually be enabled, but this relies on at least one of the external actions $Getfront$ and $Getback$ always eventually taking place. These liveness requirements can be defined in an assumption/guarantee style as follows:

$$
\begin{aligned}
& \exists n \in Nat : \\
& \Box\Diamond(Putfront(n) \vee Putback(n)) \wedge \\
& \Box\Diamond(Getfront(n) \vee Getback(n)) \Rightarrow \\
& \Box\Diamond(Enabled\ Getfront(n) \wedge Enabled\ Getback(n)) \wedge \\
& \Box\Diamond(Enabled\ Putfront(n) \wedge Enabled\ Putback(n))
\end{aligned}
$$

4 Liveness

Liveness properties define how often a particular property is satisfied. We define a number of different levels of liveness.

Definition 4.1 (Eventually Always)

Property P is *eventually always* satisifed (EA(P)) in system Π iff:

$$[\![\Pi]\!](\sigma) \Rightarrow [\![\Diamond\Box P]\!](\sigma)$$

This can also be defined as follows:

$$[\![\Pi]\!](\sigma) \Rightarrow \exists \rho \sqsubset \sigma : \rho.\tau = \sigma \wedge \forall n \in Nat : \tau[n][\![P]\!]$$

Definition 4.2 (Always Eventually)

Property P is *always eventually* satisfied (AE(P)) in system Π iff:

$$[\![\Pi]\!](\sigma) \Rightarrow [\![\Box\Diamond P]\!](\sigma)$$

This can also be defined as follows:

$$[\![\Pi]\!](\sigma) \Rightarrow \forall \rho \sqsubset \sigma : \rho.\tau = \sigma \wedge \exists n \in Nat : \tau[n][\![P]\!]$$

Definition 4.3 (Always Possibly Eventually)

Property P is *always possibly eventually* satisfied (APE(P)) in system Π iff:

$$[\![\Pi]\!](\sigma) \Rightarrow \forall \rho \sqsubset \sigma : \exists \tau : [\![\Pi]\!](\rho.\tau) \wedge [\![\Diamond P]\!](\tau)$$

This definition corresponds to the definition of *always possible* given in [7], *willingness* defined in [3] and *always possibly* given in [11].

Definition 4.4 (Possibly Always Eventually)

Property P is *possibly always eventually* satisfied (PAE(P)) in system Π iff:

$$[\![\Pi]\!](\sigma) \Rightarrow \exists \rho \sqsubset \sigma : \exists \tau : [\![\Pi]\!](\rho.\tau) \wedge [\![\Box\Diamond P]\!](\tau)$$

This definition corresponds to the definition of *may* in [6].

It is quite straightforward to prove the following:

EA(P) $\Rightarrow$ AE(P) $\Rightarrow$ APE(P) $\Rightarrow$ PAE(P)

These different levels of liveness therefore belong to the liveness domain shown in Figure 2. It is quite straightforward to prove that this is a complete lattice with a partial ordering $\sqsubseteq_L$ defined as follows:

$l_1 \sqsubseteq_L l_2$ iff $l_1 = \bot \vee l_2 \Rightarrow l_1$

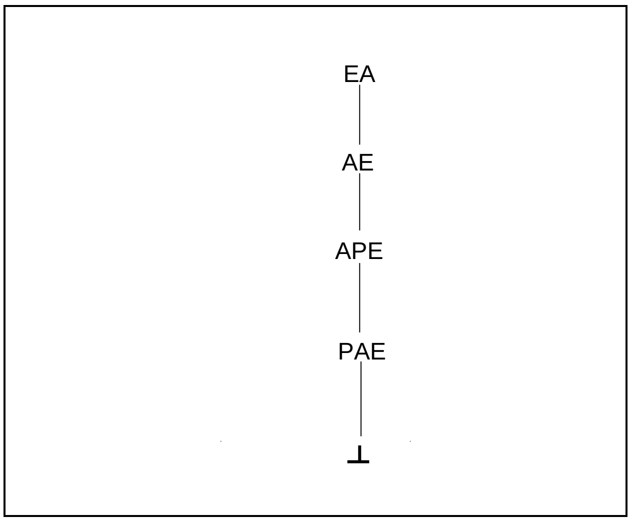

Figure 2. Liveness Domain

5 Fairness

Fairness is a means by which the liveness levels within an object can be increased in order to satisfy liveness requirements. Liveness requirements are usually of the form $\Box\Diamond P$ for some property P. We therefore define the fairness constraints required in order to satisfy these always eventually liveness requirements for different existing levels of liveness within an object.

Definition 5.1 (Weak Fairness)

A system Π is *weakly fair* with respect to action $\mathcal{A}$ (WF($\mathcal{A}$)) iff:

$$[\![\Pi]\!](\sigma) \Rightarrow [\![\Box\Diamond\mathcal{A}]\!](\sigma) \vee [\![\neg EA(Enabled\,\mathcal{A})]\!](\sigma)$$

This corresponds to the definition of *weak fairness* defined in [10].

Definition 5.2 (Strong Fairness)

A system Π is *strongly fair* with respect to action $\mathcal{A}$ (SF($\mathcal{A}$)) iff:

$$[\![\Pi]\!](\sigma) \Rightarrow [\![\Box\Diamond\mathcal{A}]\!](\sigma) \vee [\![\neg AE(Enabled\,\mathcal{A})]\!](\sigma)$$

This corresponds to the definition of *strong fairness* defined in [10].

Definition 5.3 (Hyperfairness)

A system Π is *hyperfair* with respect to action $\mathcal{A}$ (HF($\mathcal{A}$)) iff:

$$[\![\Pi]\!](\sigma) \Rightarrow [\![\Box\Diamond\mathcal{A}]\!](\sigma) \vee [\![\neg APE(Enabled\,\mathcal{A})]\!](\sigma)$$

This definition corresponds to the definition of *hyperfairness* given in [13] and *active fairness* given in [3]. It differs from the definition of *possible fairness* given in [7] in that it requires that an action is actually taken rather than just being enabled when it is always possible.

It is quite straightforward to prove the following:

HF($\mathcal{A}$) $\Rightarrow$ SF($\mathcal{A}$) $\Rightarrow$ WF($\mathcal{A}$)

These different levels of fairness belong to the following domain:

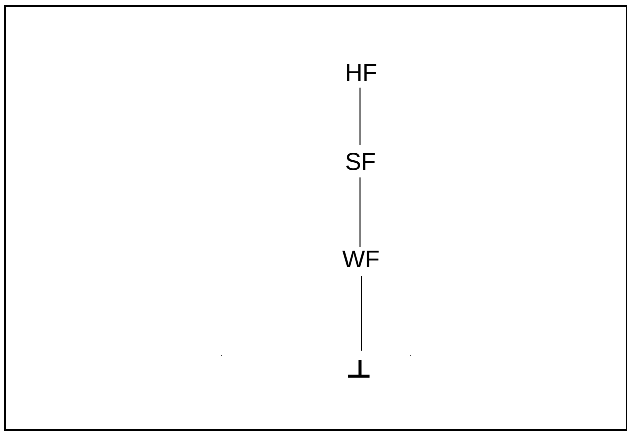

Figure 3. Fairness Domain

It is quite straightforward to prove that this is a complete lattice with a partial ordering $\sqsubseteq_F$ defined as follows:

$f_1 \sqsubseteq_F f_2$ iff $f_1 = \bot \vee f_2 \Rightarrow f_1$

It is not possible to satisfy always eventually liveness requirements using fairness constraints for any other existing levels of liveness. In particular, if an action $\mathcal{A}$ is possibly always eventually enabled (PAE($Enabled\ \mathcal{A}$)), this possibility may not exist throughout the execution of the system. It cannot therefore be ensured that the action $\mathcal{A}$ is always eventually taken without forcing or blocking some other actions earlier in the execution of the system.

We can now see how to ensure that an object satisfies its always eventually liveness requirements based on its existing levels of liveness. For example, if we require that an action $\mathcal{A}$ is always eventually taken, and we can determine that $\mathcal{A}$ is always possibly eventually enabled, then we must put a hyperfairness constraint on the action $\mathcal{A}$. The fairness constraints required for always eventually liveness requirements given different existing levels of liveness are summarised in Table 1.

Liveness	Fairness Constraints
EA($Enabled\,\mathcal{A}$)	WF($\mathcal{A}$)
AE($Enabled\,\mathcal{A}$)	SF($\mathcal{A}$)
APE($Enabled\,\mathcal{A}$)	HF($\mathcal{A}$)
PAE($Enabled\,\mathcal{A}$)	No possible constraints

Table 1. Fairness Constraints to Satisfy Always Eventually Liveness Requirements

6 Object Composition

In this section we describe the effect of object composition on the liveness properties of the composed objects. There are a number of different object composition mechanisms we could use. One such mechanism is purely parallel composition, in which there is no communication between the composed objects. This is the approach which is taken in [1] and [4]. The liveness properties of the composed object in this case are represented as the logical conjunction of the liveness properties of each individual object. Fairly simple proof obligations must be discharged to show that the liveness assumptions of the composed objects are preserved.

In general, however, we require stronger forms of object composition in which there is communication between the composed objects, which means that the liveness properties of the objects may be affected. Examples of composition mechanisms which do allow communication between the composed objects include shared variables [9, 15, 16], joint actions [8], message passing [14, 2] and sharing actions [5].

We specify object composition by the synchronisation of actions as follows:

$$O_1|[<\mathcal{A}_1^1,\mathcal{A}_1^2>,\ldots,<\mathcal{A}_n^1,\mathcal{A}_n^2>]|O_2$$

where each external action $\mathcal{A}_i^1$ in the object O_1 synchronises with the corresponding external action $\mathcal{A}_i^2$ in the object O_2. Consider the following two fair objects:

$O_1 \triangleq \exists x_1 : Init_1 \wedge \Box[Ext_1 \vee Int_1]_{x_1} \wedge F_1$
$O_2 \triangleq \exists x_2 : Init_2 \wedge \Box[Ext_2 \vee Int_2]_{x_2} \wedge F_2$

The result of composing these two objects is as follows:

$O_1|[<\mathcal{A}_1^1,\mathcal{A}_1^2>,\ldots,<\mathcal{A}_n^1,\mathcal{A}_n^2>]|O_2 \triangleq$
$\exists x_1, x_2 : Init_{12} \wedge \Box[Ext_{12} \vee Int_{12}]_{<x_1,x_2>} \wedge F_{12}$
where
$Init_{12} \triangleq Init_1 \wedge Init_2$
$Ext_1 \triangleq \mathcal{A}_1^1 \vee \ldots \vee \mathcal{A}_n^1 \vee \ldots \vee \mathcal{A}_{n+k_1}^1$
$Ext_2 \triangleq \mathcal{A}_1^2 \vee \ldots \vee \mathcal{A}_n^2 \vee \ldots \vee \mathcal{A}_{n+k_2}^2$
$Ext_{12} \triangleq \mathcal{A}_{n+1}^1 \vee \ldots \vee \mathcal{A}_{n+k_1}^1 \vee \mathcal{A}_{n+1}^2 \vee \ldots \vee \mathcal{A}_{n+k_2}^2$
$Int_{12} \triangleq Int_1 \vee Int_2 \vee \mathcal{A}_1^{12} \vee \ldots \vee \mathcal{A}_n^{12}$
$\mathcal{A}_i^{12} \triangleq \mathcal{A}_i^1 \wedge \mathcal{A}_1^2$
$F_{12} \triangleq F_1 \wedge F_2$

The external actions in the resulting composition are those external actions in the two objects which are not involved in synchronisations; synchronised actions become internal. The initial state of the resulting composition is the conjunction of the initial states of each object. Similarly, the fairness constraints on the resulting composition is the conjunction of the fairness constraints on each object. The synchronised actions are also the conjunction of the individual actions involved in the synchronisation. For example, consider the composition of two double-ended queues to produce another double-ended queue as shown in Figure 4.

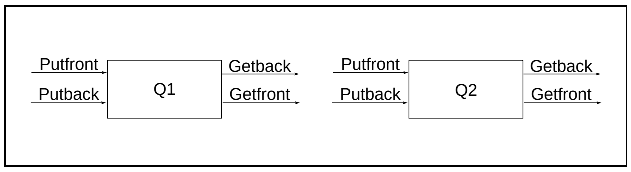

Figure 4. Composed Double-Ended Queues

This composition would be represented as follows:

$Q_1|[<Getback, Putfront>, <Getfront, Putback>]|Q_2$

The TLA specification resulting from this specification is as follows:

$Addfront(n) \triangleq \wedge\ Len(Q1.contents) < 100$
$\qquad\wedge\ Q1.contents' =< n > \circ Q1.contents$

$Addback(n) \triangleq \wedge\ Len(Q2.contents) < 100$
$\qquad\wedge\ Q2.contents' = Q2.contents \circ < n >$

$Getfront(n) \triangleq \wedge\ Len(Q1.contents) > 0$
$\qquad\wedge\ Q1.contents =< n > \circ Q1.contents'$

$Getback(n) \triangleq \wedge\ Len(Q2.contents) > 0$
$\qquad\wedge\ Q2.contents = Q2.contents' \circ < n >$

$Internal1 \triangleq \wedge\ Len(Q1.contents) > 0$
$\qquad\wedge\ Q1.contents = Q1.contents' \circ < n >$

$$\begin{aligned}
&\land Len(Q2.contents) < 100\\
&\land Q2.contents' =< n > \circ Q2.contents
\end{aligned}$$

$$\begin{aligned}
Internal2 \triangleq\ &\land Len(Q2.contents) > 0\\
&\land Q2.contents =< n > \circ Q2.contents'\\
&\land Len(Q1.contents) < 100\\
&\land Q1.contents' = Q1.contents \circ < n >
\end{aligned}$$

$$\begin{aligned}
Step \triangleq \exists n \in Nat :\ &\lor Addfront(n)\\
&\lor Addback(n)\\
&\lor Getfront(n)\\
&\lor Getback(n)\\
&\lor Internal1\\
&\lor Internal2
\end{aligned}$$

$$Init \triangleq Q1.contents =<> \land Q2.contents =<>$$

$$\begin{aligned}
Spec \triangleq\ &\exists Q1.contents, Q2.contents \in Seq(Nat) :\\
&\quad Init \land \Box[Step]_{<Q1.contents, Q2.contents>}
\end{aligned}$$

This composed object will operate normally as a double-ended queue, provided that the environmental liveness requirements for each of the composed objects are upheld. However, it is possible that the actions on which the two objects synchronise will cause their environmental liveness requirements to be broken. For example, consider the synchronisation between *Q1.Getback* and *Q2.Putfront*. If the environmental liveness requirements for both these objects were satisfied, then both of these actions would be always eventually enabled, but it is possible that they will never actually be enabled at the same time. The internal action *Internal1* may therefore never actually be taken. Similarly, the internal action *Internal2* may never be taken. The environmental liveness requirements for each object have therefore been broken.

Liveness requirements which are broken due to synchronisation may be resolved by placing additional fairness constraints on the synchronised actions. In order to determine the fairness constraint which must placed on synchronised actions, we need to determine their level of liveness. A synchronised action is enabled only if the individual actions involved in the synchronisation are enabled at the same time within each object. We can therefore determine the liveness of a synchronised action from the liveness of the individual actions involved in the synchronisation. We might expect that the liveness of a synchronised action is given by the least upper bound of the liveness of the individual actions involved in the synchronisation within the liveness domain defined in Figure 2. However, the example of the double-ended queues shows that this is not necessarily the case. Even if the actions involved in the synchronisation are always eventually enabled, they may never actually be enabled at the same time. The synchronised action in this case will only be always possibly eventually enabled. The liveness of a synchronised action is defined as follows:

Definition 6.1 (Liveness of Synchronised Actions)

The liveness of a synchronised action can be determined from the liveness of the individual actions involved in the synchronisation according to the following table:

	Object 1			
Object 2	EA	AE	APE	PAE
EA	EA	AE	APE	PAE
AE	AE	APE	APE	PAE
APE	APE	APE	APE	PAE
PAE	PAE	PAE	PAE	PAE

Table 2. Liveness of Synchronised Actions

Each of the entries in this table are proved on an individual basis. The details of these proofs are not given here.

Now that the liveness of synchronised actions can be determined, the fairness constraints which must be placed on them to satisfy liveness requirements can be determined. For example, in the case of the composed double-ended queues we can determine that the synchronised actions *Internal1* and *Internal2* are always possibly eventually enabled. Hyperfairness constraints must therefore be placed on these actions to ensure that the liveness requirements of the composed objects are satisfied.

7 Politeness

Politeness is the property which the environment of an object must have to uphold the environmental liveness requirements of that object. When objects are composed, we need to show that the environmental liveness requirements of each object are upheld within the resulting composed object. We can classify these objects as being *independent*, *perfect friends*, *friends*, *politicians* or *enemies* depending on their level of politeness. We now give a definition of these different levels of politeness.

Definition 7.1 (Independent) *Two composed objects are defined to be independent if they do not synchronise on any actions.*

This will only be the case if the two objects are composed using purely parallel composition.

Definition 7.2 (Perfect Friends) *Two composed objects are defined to be perfect friends if they do synchronise on actions, but the environmental liveness requirements of the composed objects are not affected.*

This will only be the case if all the actions within each object which are involved in synchronisations are eventually always enabled. These objects will be perfect friends because the liveness of the synchronised actions will not be affected. Perfect friends can therefore be defined more formally as follows. If objects O_1 and O_2 are composed as follows:

$O_1|[<\mathcal{A}_1^1, \mathcal{A}_1^2>, \ldots, <\mathcal{A}_n^1, \mathcal{A}_n^2>]|O_2$

Then O_1 and O_2 are perfect friends iff:

$\forall \mathcal{A} \in \{\mathcal{A}_1^1 \ldots \mathcal{A}_n^1, \mathcal{A}_1^2 \ldots \mathcal{A}_n^2\}$: EA($Enabled\,\mathcal{A}$)

Definition 7.3 (Friends) *Two composed objects are defined to be friends if they synchronise on actions, but the environmental liveness requirements of the composed objects are affected, so additional fairness constraints are required on the synchronised actions to restore the environmental liveness requirements of the composed objects.*

This will be the case if the two objects are not perfect friends, but all the actions within each object which are involved in synchronisations are at least always possibly eventually enabled. These objects will be friends because the environmental liveness requirements of the composed objects can be restored by placing hyperfairness constraints on the synchronised actions. For the double-ended queues example, we must ensure that all the synchronised actions are always eventually taken. We can do this by placing a hyperfairness constraint on them (since they are always possibly eventually enabled). Friends can therefore be defined more formally as follows. If objects O_1 and O_2 are composed as follows.

$O_1|[<\mathcal{A}_1^1, \mathcal{A}_1^2>, \ldots, <\mathcal{A}_n^1, \mathcal{A}_n^2>]|O_2$

Then O_1 and O_2 are friends iff:

$\forall \mathcal{A} \in \{\mathcal{A}_1^1 \ldots \mathcal{A}_n^1, \mathcal{A}_1^2 \ldots \mathcal{A}_n^2\}$: APE($Enabled\,\mathcal{A}$)
$\wedge\ \exists \mathcal{A} \in \{\mathcal{A}_1^1 \ldots \mathcal{A}_n^1, \mathcal{A}_1^2 \ldots \mathcal{A}_n^2\}$: $\neg$ EA($Enabled\,\mathcal{A}$)

Definition 7.4 (Politicians) *Two composed objects are defined to be politicians if they synchronise on actions, but the environmental liveness requirements of the composed objects are affected, and these requirements cannot be restored by just adding fairness constraints; some additional resolution mechanism (such as the blocking/forcing of actions) is required.*

This will be the case if the two objects are not friends, but all the actions within each object which are involved in synchronisations are at least possibly always eventually enabled. These objects will be politicians because the environmental liveness requirements of the composed objects can be restored by the blocking/forcing of actions. Politicians can therefore be defined more formally as follows. If objects O_1 and O_2 are composed as follows:

$O_1|[<\mathcal{A}_1^1, \mathcal{A}_1^2>, \ldots, <\mathcal{A}_n^1, \mathcal{A}_n^2>]|O_2$

Then O_1 and O_2 are politicians iff:

$\forall \mathcal{A} \in \{\mathcal{A}_1^1 \ldots \mathcal{A}_n^1, \mathcal{A}_1^2 \ldots \mathcal{A}_n^2\}$: PAE($Enabled\,\mathcal{A}$)
$\wedge\ \exists \mathcal{A} \in \{\mathcal{A}_1^1 \ldots \mathcal{A}_n^1, \mathcal{A}_1^2 \ldots \mathcal{A}_n^2\}$: $\neg$ APE($Enabled\,\mathcal{A}$)

Definition 7.5 (Enemies) *Two composed objects are defined to be enemies if they synchronise on actions, but the environmental liveness requirements of the composed objects are affected, and these requirements cannot be restored by adding fairness constraints or by using any additional resolution mechanism.*

This will be the case if at least one action within the two objects which are involved in synchronisations may never be enabled. Enemies can therefore be defined more formally as follows. If objects O_1 and O_2 are composed as follows:

$O_1|[<\mathcal{A}_1^1, \mathcal{A}_1^2>, \ldots, <\mathcal{A}_n^1, \mathcal{A}_n^2>]|O_2$

Then O_1 and O_2 are enemies iff:

$\exists \mathcal{A} \in \{\mathcal{A}_1^1 \ldots \mathcal{A}_n^1, \mathcal{A}_1^2 \ldots \mathcal{A}_n^2\}$: $\neg$ PAE($Enabled\,\mathcal{A}$)

8 Conclusions

In this paper, the effect of object composition on the liveness properties of the objects being composed has been studied under the assumption that the preservation of liveness properties is hard to achieve. It was not considered sufficient to produce proof obligations which must be discharged in order to show that liveness properties are preserved; it must also be shown how to resolve interactions in situations where they are not preserved.

A number of different levels of liveness were defined, and a number of corresponding fairness constraints were also defined which can be used to ensure that liveness levels within an object can be increased to satisfy their liveness requirements. It was shown how the liveness properties of objects interact when they are composed with other objects, and the degree to which they interact was defined within a hierarchy of politeness. It was then shown how to resolve liveness interactions for one of these levels of politeness.

Finally, we have not considered the composition of objects by inheritance, or the refinement of objects within our framework. These are the subject of further research.

References

[1] M. Abadi and L. Lamport. Conjoining Specifi cations. *ACM Trans. Prog. Lang. Syst.*, 17(3):507–533, May 1995.

[2] J. Bahsoun, S. Merz, and C. Servières. Modular description and verifi cation of concurrent objects. In *0bject-Based Parallel and Distributed Computation*, volume 1107 of *Lecture Notes in Computer Science*, 1996.

[3] N. Barreiro, J. Fiadeiro, and T. Maibaum. Politeness in Object Societies. In R. Wieringa and R. Feenstra, editors, *Information Systems: Correctness and Reusability*, pages 119–134. World Scientifi c Publishing Company, 1995.

[4] E. Canver and F. von Henke. Formal development of object-based systems in a temporal logic setting. In *Formal Methods for Open Object-Based Distributed Systems*, pages 419–436. Kluwer Academic Publishers, February 1999.

[5] J. Fiadeiro and T. Maibaum. Sometimes "Tomorrow" is "Sometime": Action Refi nement in a Temporal Logic of Objects. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic*. Springer Verlag, 1994.

[6] J. Gibson and D. M´ery. Always and eventually in object requirements. In *ROOM2*, Bradford, 1998.

[7] J. Gibson and D. M´ery. Fair objects. In *Object Technology 98*, Oxford, 1998.

[8] H.-M. Järvinen and R. Kurki-Suonio. DisCo Specifi cation Language: Marriage of Actions and Objects. In *11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE Computer Society Press, 1991.

[9] C. Jones. Tentative Steps Towards a Development Method for Interfering Programs. *ACM Trans. Prog. Lang. Syst.*, 5(4):596–619, Oct. 1983.

[10] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[11] L. Lamport. Proving Possibility Properties. Research Report SRC-137, Digitial Systems Research Center, 1995.

[12] L. Lamport. The Module Structure of TLA+. SRC Technical Note 1996-002, Digitial Systems Research Center, 1996.

[13] L. Lamport. Fairness and Hyperfairness. Research Report SRC-152, Digitial Systems Research Center, 1998.

[14] J. Misra and M. Chandy. Proofs of Networks of Processes. *IEEE SE*, 7(4):417–426, 1981.

[15] C. Stirling. A Generalization of Owicki-Gries's Hoare Logic For a Concurrent While Language. *Science of Computer Programming*, 58:347–359, 1988.

[16] Q. Xu, W.-P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.