

Fair Objects

Paul Gibson¹ and Dominique Méry¹

Université Henri Poincaré (Nancy I)
UMR 7503 LORIA, Campus Scientifique,
BP 239, 54506 VANDOEUVRE-LES-NANCY, FRANCE.
Phone: +33 3 83 59 20 14. Fax: +33 3 83 41 30 79.
Email: (gibson@loria.fr) (mery@loria.fr). WWW: <http://www.loria.fr/~mery>.

Abstract. The temporal logic of actions (TLA) provides operators to express liveness requirements in an abstract specification model. TLA does not, however, provide high level composition mechanisms which are essential for synthesising and analysing complex behaviour. Contrastingly, the object oriented paradigm has proven itself in the development of structured specifications. However, most, if not all, of the object oriented formalisms are based on the specification of safety properties and, as such, they do not provide an adequate means of expressing liveness conditions. This paper examines how we combine temporal semantics and object oriented concepts in a complementary fashion. High level re-usable concepts are formalised as different kinds of *fair objects*. The object oriented semantics aid validation and customer communication, whilst the TLA semantics provide a means of formally verifying liveness requirements. The fairness concepts are founded on the notion of objects as servers which may have multiple (concurrent) clients. Some simple telephone feature specifications illustrate the practical application of our *fair object* semantics.

1 Introduction

We believe a complementary integration of object oriented concepts and TLA[1] within one formal framework is an aid to requirements modelling. Our main goal is to draw together the object oriented, concurrency and fairness concepts into one formalism. Combining the different semantic frameworks is not an easy task. We are currently following two different approaches which, we hope, will meet in the middle. Firstly, we are examining the means by which we can improve Lamport's proposals for structural mechanisms in TLA[2], which will reflect the object oriented semantics of classification, composition, subclassing and polymorphism. Secondly, we are extending the object oriented semantics to include the notion of different forms of *fair objects*. This paper reports on this second line of work. Namely, we report on the different forms of *fair object* which act as high level reusable components during requirements modelling. Throughout this paper we do not report on the formal integration of our two different semantic models. Rather, specifications are given pairwise and the intuitive relationship between each model in a pair is informally, yet rigorously, explained.

1.1 Objects

Object oriented methods encompass a set of techniques which have been, and will continue to be, applied in the successful production of complex software systems. The methods are based on the simple mathematical models of abstraction and classification. We adopt a simple object-labelled state transition system semantics (O-LSTS) which

regards an object as a state transition machine [4]. The true advantage of the O-LSTS approach is in the initial requirements modelling phase of development. It improves the communication with the customer and aids in synthesis, analysis and validation. However, we have found the inability to express liveness requirements a main weakness when applying the O-LSTS approach. Without liveness we can specify only what cannot happen (i.e. safety properties) rather than what must happen. Furthermore, without temporal semantics based on liveness, the nondeterminism in a system can be specified only at one level of abstraction: namely that of an internal choice of events. This can lead, as we shall later see, to many problems in development.

1.2 Temporal Logic and Liveness

Consider the specification of a shared database. This database must handle multiple, parallel requests from clients. The order in which these requests are processed is required to be nondeterministic. This is easily specified without liveness. However, if the requirements are now refined to state that every request must be eventually served (this is a fairness requirement which we cannot express in a safety-only semantic framework). Our only choice is to *over-specify* the requirement by defining how this fairness is to be achieved (for example, by explicitly queueing the requests). This is bad because we are enforcing implementation decisions at the requirements level.

TLA [1] provides a simple and effective means of expressing fairness properties. The semantics incorporate the notions of *always* (represented by the \square operator) and *eventually* (represented by the \diamond operator). Using these, we can specify different categories of fairness within the object oriented framework. The ability to model nondeterminism at different levels of abstraction is the key to TLA's utility in requirements modelling. Unfortunately, TLA does not provide the means for easily constructing and validating initial customer requirements. By combining TLA and object oriented semantics we can alleviate these problems.

1.3 Telephone Feature Specification

Features are observable behaviour and are therefore a requirements specification problem [5]. We concentrate on the domain of telephone features [6,7]. The feature interaction problem is stated simply, and informally, as follows: A *feature interaction* is a situation in which system behaviour (specified as some set of features) does not as a whole satisfy each of its component features individually. Most feature interaction problems can be (and should be) resolved at the requirements capture stage of development[8]. The telephone feature examples in section 4 are taken from a large list of specifications which we have developed using our *fair object* concepts. Telephone feature specification is well suited to our semantic approach because it requires a high degree of structuring to cope with the highly compositional and incremental nature of such systems; and there is a clear need for fairness requirements [9].

2 Semantic Framework

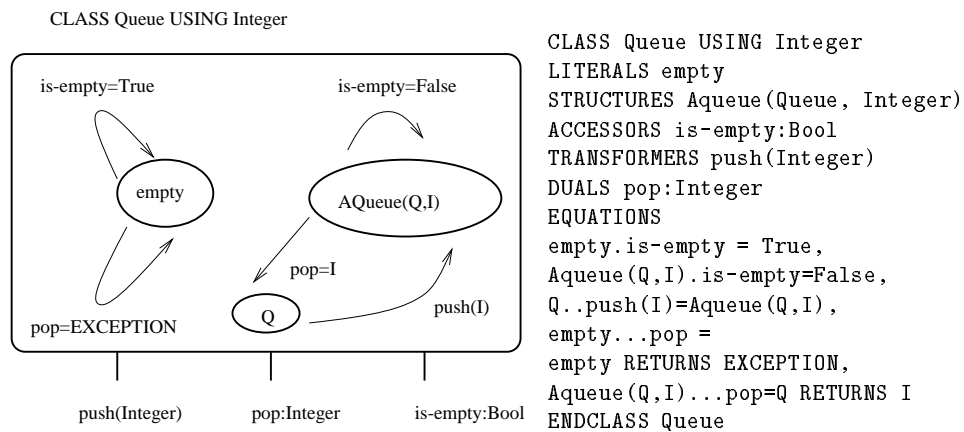
2.1 Objects as State Machines: Different Views

Labelled state transition systems are often used to provide executable models during analysis, design and implementation stages of software development [10–12]. In particular, such models are found in the classic analysis and design methods of [13–15].

However, a major problem with state models is that it can be difficult to provide a good system (de)composition when the underlying state and state transitions are not easily conceptualised. The object oriented paradigm provides a natural solution to this problem. By equating the notion of class with the state transition system model and allowing the state of one class to be defined as a composition of states of other classes, the O-LSTS approach provides a means of specifying such models in a constructive fashion.

The O-LSTS semantics also permit us to view objects at different levels of abstraction. Firstly, using an abstract data type (ADT) we can specify the functionality of an object at a level of abstraction suitable for requirements capture[16]. Secondly, we can transform our ADT requirements into a parameterised process algebra (LOTOS[17,18]) specification for the design stage. Finally, as we approach an implementation environment, we can view the objects in our designs as clients and servers in a distributed, concurrent network. At each of these levels of abstraction we provide a means of incorporating fairness requirements.

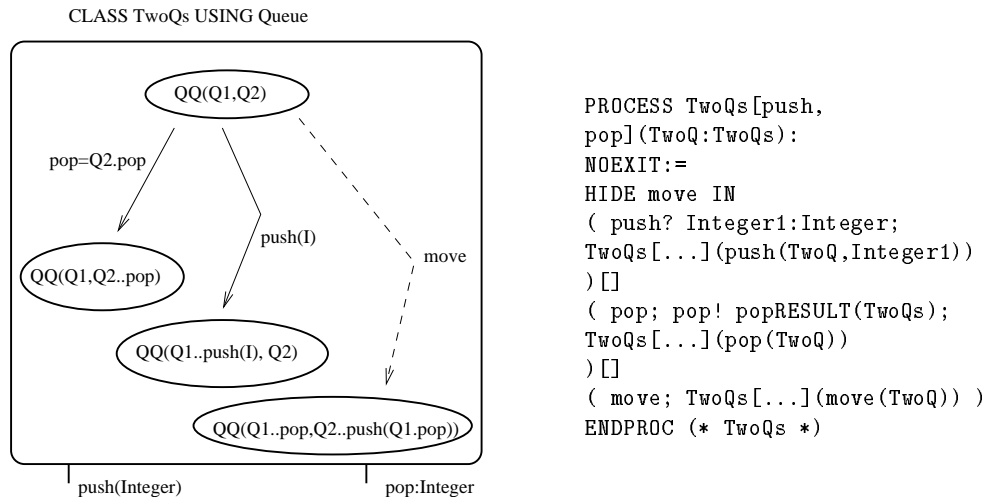
ADT View — The simplest way to introduce the ADT view is through a *standard* example: A **Queue** of **Integers** is specified using the OO ACT ONE specification language from [3]. An equivalent graphical representation of the state transition system is also given.



The **Queue** uses a predefined class **Integer** (which itself uses **Bool**) The literal and structure members define all the possible (states of the) objects in the class. There is one non-structured literal value **empty**. All other elements are structured from two components, namely a **Queue** and an **Integer**, using the **AQueue** operator. The class interface is defined by the three different sets of services offered: a **transformer** changes the internal state of an object, an **accessor** returns some value without changing internal state, a **dual** returns a value and may also change the internal state. The services may be parametrised (e.g. **push**) to represent the input data passed to the server object when a service is requested. The equations are used to define a semantics for the interface services. The syntax (including the *dot notation*) should be easy to follow: *one dot* defines the value returned when a service is completed, *two dots* defines the new state of an object after a service is fulfilled, and *three dots* defines both these values. It should be noted that every class has, by default, an implicit **EXCEPTION** value which can be used in cases like the popping of an element from an empty queue. Furthermore, variables which are not typed explicitly may have their types inferred

where there is no ambiguity. We also have tools for verifying the completeness and consistency of such specifications.

Process Algebra View — Consider a system made up of two **Queues**, as specified above, whose behaviour is illustrated by the O-LSTS diagram for class **TwoQs**. (The corresponding OO ACT ONE code can be easily deduced from the diagram: the only new construct is the *dotted move* transition which represents an internal, nondeterministic state transformer.) A corresponding full LOTOS design¹ is found to the right of the diagram. The ADT part of the LOTOS specification, not shown, is generated automatically from the OO ACT ONE specification, and is used to parameterise the process definition for the corresponding behaviour. In this case, type **TwoQs** parameterises the behaviour of process **TwoQs**.



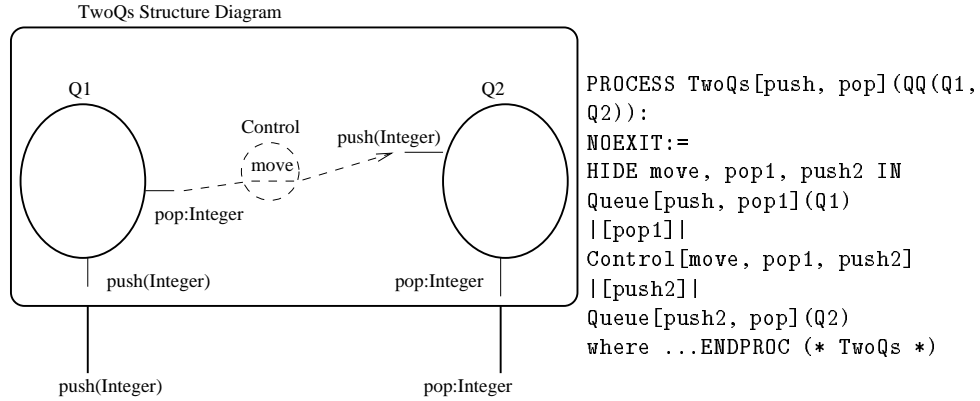
The **TwoQs** provides two services: **push** (which pushes elements onto the first queue component) and **pop** (which pops elements from the second queue component); and the internal state transition **move** transfers elements from the first queue onto the second queue. The full LOTOS design moves us a step closer to an implementation because our specification is now modelled as a process with which we can communicate. This particular LOTOS design, chosen for its simplicity, specifies that a *remote procedure call* protocol² is used for communication with the **TwoQs** process. The **push** operation is carried out synchronously between the object and its environment. The **pop** operator requires some result to be returned and we model the communication of the result as an event different from the service request. The **move** operation is hidden from the environment of the **TwoQs** process: as such, the movement of elements between the two queues cannot be determined by the **TwoQs** client(s).

It is now possible, using a pre-defined correctness preserving transformation, to re-use the compositional structure found in the OO ACT ONE specification in a structurally equivalent compositional LOTOS specification. This results in an equivalent specification with two **Queue** processes synchronising on an internal **move** event. We

¹ For simplicity, we use a shorthand '[...]' to represent a gate list which is the same as that found in the process definition header.

² Other types of protocol can also be generated automatically.

illustrate this below, in a partial specification together with an equivalent O-LSTS structure diagram:



Each of the component **Queue** processes is generated directly from the ADT specification. The names of the external services of these components are indiced to avoid name clashes, and all synchronisation between the two components is controlled by a **Control** process. This **Control** is generated through analysis of the equations defined in the **TwoQs** ADT. There will (nearly) always be a need for such a control process: we consider it as a refinement of the actions which are shared between the component parts. In this example, we note that **push** and **pop** do not require co-operation between the two components and so these can be routed directly to the internal components.

Client-Server View — The type of LOTOS design seen above is quite close to the type of *client-server* model that is found in many reference models for software development, see [19,20], for example. We can say that the environment of a **TwoQs** process is its client. Now let us consider a *liveness* property which we would reasonably require such a system to fulfil. The nondeterministic **move** operation cannot be guaranteed to be carried out when we specify only *safety* properties. We may require that if an element is pushed onto the first queue, then it will eventually be moved to the second queue. We do not wish to specify how this happens, only that it does. This is the essence of abstraction with regards to the nondeterminism in our system: we need to be able to specify *fairness* at the ‘class level of abstraction’.

Furthermore, we need to consider what happens when a server has multiple (concurrent) clients. If an object in the server’s environment requests a **push** how can we be sure that it will be carried out even though, in this case, it is always enabled? The problem is as follows: if the server is shared between other clients then how do we guarantee that one client’s requests will eventually be carried out? Certainly, we could specify some sort of queueing protocol, for example. However, again we believe that at this stage of design we do not want to impose such implementation decisions. A *fair object* should guarantee, wherever possible, the eventuality of meeting clients’ requests in a concurrent environment. Using TLA, we shall show how different types of *fair objects* can be used to fulfil such eventuality requirements.

2.2 Temporal Logic

Introducing TLA — TLA is a linear temporal logic introduced by Lamport [1] and based on the action-as-relation principle. A system is considered as a set of actions,

namely a logical disjunction of predicates relating values of variables before the activation of an action and values of variables after the activation of an action; a system is modeled as a set of traces over a set of states. The specifier may decide to ignore traces that do not satisfy a scheduling policy, such as strong or weak fairness; and temporal operators such as \square (Always) or \diamond (Eventually) are combined to express these assumptions over the set of traces.

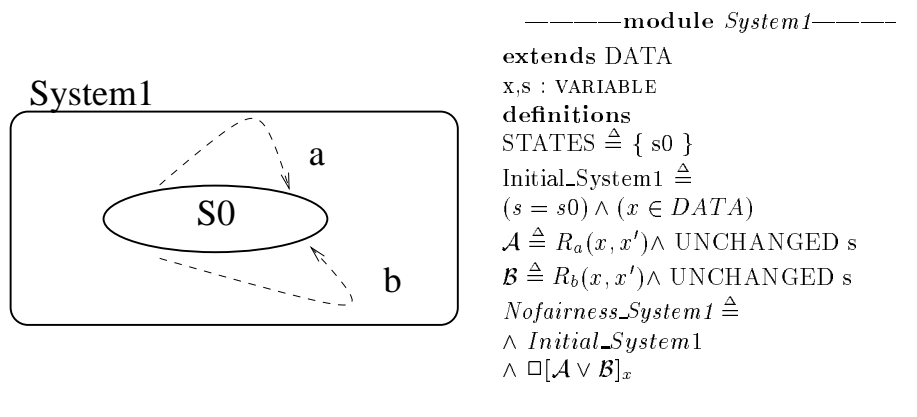
The meaning $\llbracket I \rrbracket$ of a linear temporal-logic formula I is a Boolean-valued function on behaviors. We say that the behavior σ satisfies I iff $\llbracket I \rrbracket(\sigma)$ equals TRUE. Formula I is valid, written $\models I$, iff every behavior satisfies I . To use temporal logic to specify (a mathematical model of) a system, we consider states to represent possible system states and events to represent possible system actions, so a behavior represents a conceivable execution of a system. A system is specified by a formula I that is satisfied by precisely those behaviors that represent a legal system execution.

The power of TLA with respect to verification is its simplicity. One model is verified against another simply by logical implication. The means of verification are built into the language semantics.

Liveness and Safety — A safety property expresses that something bad will never happen. More precisely, a safety property is a formula on traces that is satisfied by an infinite behavior σ if, and only if, it is satisfied by every prefix of σ . It is finitely refutable. Alpern and Schneider have shown that a safety property is a closed set in an adequate topology[21]. Intuitively, a safety property I constrains only the finite behavior of a system—any behavior that fails to satisfy I fails at some specific instant.

A liveness property states that something good will eventually happen. The two most commonly seen liveness properties, within the TLA framework, are weak fairness and strong fairness. We examine these below, where we incorporate them into our *fair object* framework.

Weak fairness — Consider the simple O-LSTS for **System1**. (We represent only internal actions and abstract away from the state values that play no role in deciding which internal actions are enabled. In this way the example is trivial, but it does illustrate the need for fairness.) The system permits traces **a**; **a**; **a**; ... and **b**; **b**; **b**; ... In other words, we cannot guarantee that either **a** or **b** will ever be performed. The corresponding temporal logic specification (based on TLA syntax) is also given.



The specification of *Nofairness_System1* guarantees, through the always operator \Box , the eventuality of some action being carried out. It does not guarantee either of the actions individually: for that we need the notion of *weak fairness*. Weak fairness states that if an action is continually enabled then it will be eventually carried out. This is defined below, together with other useful concepts:

Definition 1. Fundamental TLA Concepts

p in the next state: $p' \triangleq p(\forall v : v'/v)$

Eventually p in a linear temporal model: $\diamond F \triangleq \neg\Box\neg F$

Either action \mathcal{A} or a stuttering step: $[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f' = f)$

F Leads to G: $F \rightsquigarrow G \triangleq \Box(F \Rightarrow \diamond G)$

An \mathcal{A} step without stuttering: $\langle \mathcal{A} \rangle_f \triangleq \mathcal{A} \wedge (f' \neq f)$

Unchanged: $\text{UNCHANGED } f \triangleq f' = f$

Definition 2. Weak Fairness

Weak fairness: $\text{WF}_f \mathcal{A} \triangleq \Box\diamond \langle \mathcal{A} \rangle_f \vee \Box\diamond\neg\text{ENABLED } \langle \mathcal{A} \rangle_f$

We define weak fairness on an object (system), written $\text{WF}(\text{system})$ to be the system together with weak fairness requirements on all the internal events in its alphabet. Using the TLA theorem prover TLP [22], we can now prove that neither **a** nor **b** actions in $\text{WF}(\text{System1})$ will be continually refused in order to be carried out the other (see *Property1* in the specification).

—————**module** *WF_System1*—————

extends System1

definitions

$\text{WF_System1} \triangleq \wedge \text{Initial_System1}$
 $\wedge \Box[\mathcal{A} \vee \mathcal{B}]_x$
 $\wedge \text{WF}\langle x, s \rangle \mathcal{A}$
 $\wedge \text{WF}\langle x, s \rangle \mathcal{B}$

theorems

$\text{Property1} \triangleq \text{WF_System1} \Rightarrow \Box\diamond \langle \mathcal{A} \rangle_{\langle x, s \rangle} \wedge \Box\diamond \langle \mathcal{B} \rangle_{\langle x, s \rangle}$

assumptions

$\text{Assumption_}\mathcal{A} \triangleq \forall x \in \text{DATA}, \exists x' \in \text{DATA} : R_a(x, x')$
 $\text{Assumption_}\mathcal{B} \triangleq \forall x \in \text{DATA}, \exists x' \in \text{DATA} : R_b(x, x')$

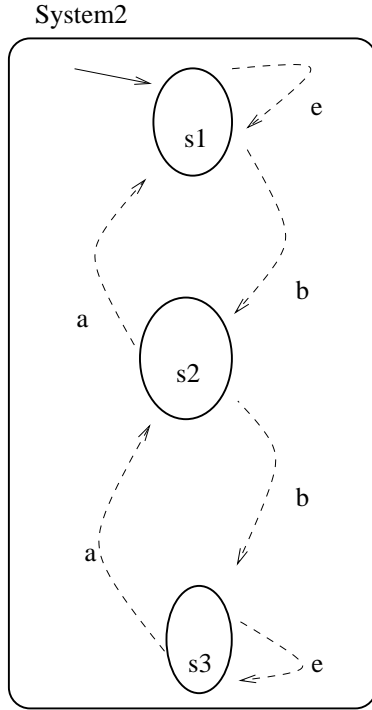
Now, in the specification of the **TwoQs** system we can use weak fairness to guarantee that the **move** operation cannot be continually ignored. Thus, specifying $\text{WF}(\text{TwoQs})$ guarantees that we shall always eventually be able to **pop** off an element (from the second queue component) provided we have already **pushed** an element (onto the first queue component). Using TLP, we can now prove that $\text{WF}(\text{TwoQs}) \Rightarrow \text{Queue}$, i.e. the system of two queues is an implementation of a simple queue.

Strong fairness — Weak fairness guarantees the eventual execution of an action when that action is continually enabled. If the action is not continually enabled, but is enabled an infinite number of times, then weak fairness does not guarantee its execution. This is the role of strong fairness. Consider the simple O-LSTS for **System2**. The system permits the trace **a;b;a;b;a;b;...** We may require that event **e** always

eventually happens. However, stating weak fairness on **System2** (i.e. weak fairness on all its actions) does not guarantee the execution of any **e** events. For this we require strong fairness. By stating strong fairness, as defined below, on action \mathcal{E} in **System2**, we can prove its eventual execution:

Definition 3. Strong Fairness

$$SF_f \mathcal{A} \triangleq \Box \Diamond \langle \mathcal{A} \rangle_f \vee \Diamond \Box \neg \text{ENABLED} \langle \mathcal{A} \rangle_f$$



-----module *System2*-----

extends DATA
 x, s : VARIABLE
definitions
STATES $\triangleq \{ s1, s2, s3 \}$
Initial_System2 $\triangleq (s = s1) \wedge (x \in DATA)$
 $\mathcal{A} \triangleq$
 $\vee R_a(x, x') \wedge (s = s2) \wedge (s' = s1)$
 $\vee R_a(x, x') \wedge (s = s3) \wedge (s' = s2)$
 $\mathcal{B} \triangleq$
 $\vee R_b(x, x') \wedge (s = s1) \wedge (s' = s2)$
 $\vee R_b(x, x') \wedge (s = s2) \wedge (s' = s3)$
 $\mathcal{E} \triangleq$
 $R_e(x, x') \wedge ((s = s1) \vee (s = s3))$
 $\wedge \text{UNCHANGED } s$
SF_System2 $\triangleq \wedge \text{Initial_System2}$
 $\wedge \Box [\mathcal{A} \vee \mathcal{B} \vee \mathcal{E}]_x$
 $\wedge SF \langle x, s \rangle \mathcal{A}$
 $\wedge SF \langle x, s \rangle \mathcal{B}$
 $\wedge SF \langle x, s \rangle \mathcal{E}$

theorems
P1 \triangleq
SF_System2 $\Rightarrow \Box \Diamond \langle \mathcal{E} \rangle_{\langle x, s \rangle}$

assumptions
Assumption_ $\mathcal{A} \triangleq$
 $\forall x \in DATA, \exists x' \in DATA : R_a(x, x')$
Assumption_ $\mathcal{B} \triangleq$
 $\forall x \in DATA, \exists x' \in DATA : R_b(x, x')$
Assumption_ $\mathcal{E} \triangleq$
 $\forall x \in DATA, \exists x' \in DATA : R_e(x, x')$

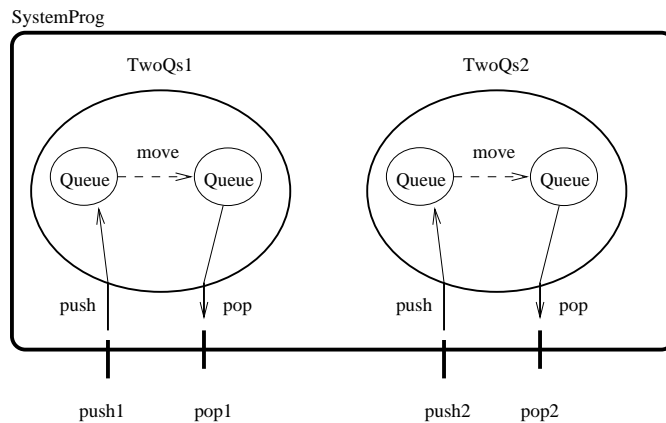
There is no need for strong fairness in the **TwoQs** system to guarantee the movement of elements. However, imagine a similar system where a **move** is enabled only directly after a **push** event. In this instance, we would need to define strong fairness on this system to guarantee the movement of elements between the two queues. We define $SF(\text{system})$ as the **system** in which all its internal actions have the strong fairness requirement. We note that SF is more commonly found than WF: it is not often that internal actions are continually enabled since in most specifications there are intermediate states, like **s2** in **System2**, where a required action is not enabled but is sure to be enabled in the next state.

3 Fair Objects

The goal of our research is to identify high-level fairness concepts within our object oriented framework. The O-LSTS specifications can then be extended to include fairness requirements, which are to be reasoned about using TLA. *Weakly fair objects* and *strongly fair objects* provide us with two such concepts; five additional high-level concepts are described below. Each of these has played an important role in the development of telephone feature specifications.

3.1 Progression

The notion of progression arises from the way in which concurrent processes are modelled through an interleaving of events. We can "view" such interleaving as though there is a scheduler which randomly chooses which process to be executed at any particular time. In such systems we wish to specify that each of the component processes is *fairly scheduled*. Consider the simple example in the diagram below, where we consider concurrent **TwoQs** component objects which do not need to communicate within the system in which they are found:



In **SystemProg** we can not guarantee the internal movement of elements in either of the two components, even if we specify the component objects to be fair (strong or weak). The reasoning is simple: the interleaving semantics, of the process algebra, would allow either one of the two processes never to be scheduled and thus the internal **moves** may never be enabled. Thus, we require some means of saying that the scheduling in **SystemProg** is *fair*. Furthermore, we want to be able to do this without having to change the specifications of the two components. *Progression* corresponds to this notion. In this simple example, we can see that $\text{Prog}(\text{SystemProg})$ could be defined as strong fairness on the action $(\text{move}_{\text{TwoQs1}} \vee \text{move}_{\text{TwoQs2}})$. This can be easily formalised, and generated automatically, in TLA.

A more complex situation occurs if there is synchronisation between two, or more, components of a system. In such a case, we require strong fairness on the synchronising action(s) and progression in the individual components:

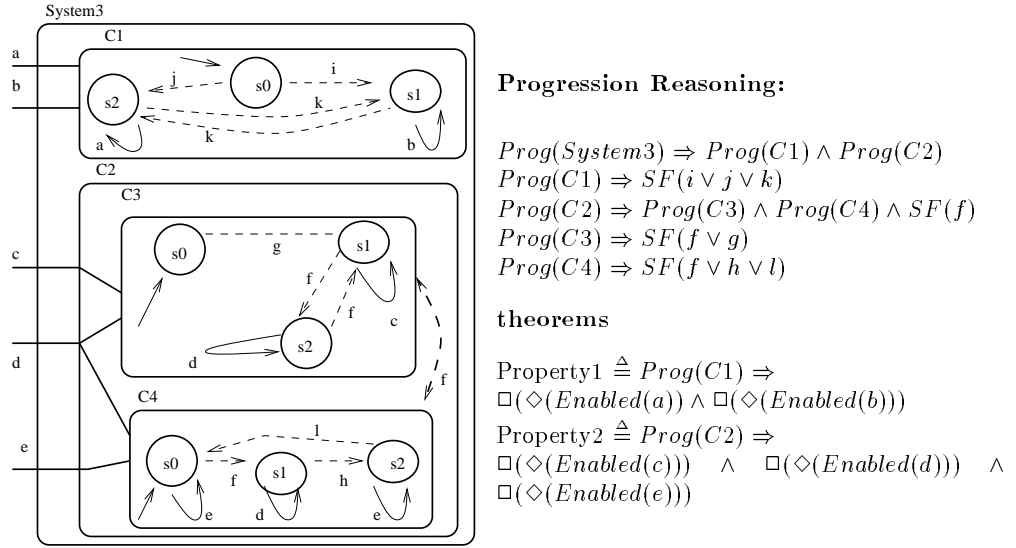
Definition 4. Progression

A process P is said to (strongly) progress, written $\text{Prog}(P)$, when:

- If P has no component processes then we require $SF(\bigvee_{i \in I} i)$, where I is the internal action set of P .
- If P has components $C1$ and $C2$ then each of these components progress and $SF(\bigvee_{i \in J} i)$, where J is the set of internal actions on which $C1$ and $C2$ synchronise.

This definition is easily generalised to processes with more than two components.

A weaker form of progression can be defined by placing only WF constraints on the internal action sets. By default, when talking about progression, we mean strong progression. The utility of progression is illustrated by **System3**, where we see that progression is needed to guarantee the *eventuality obligations* (see 3.5): $\forall x \in \{a, b, c, d, e\}: \Box \Diamond Enabled(x)$. Thus, using progression, we can prove that the environment of **System3** will never have to wait indefinitely for a requested service to be enabled.



3.2 Possible Fairness

Nondeterminism often gives rise to systems in which it is always possible for an action to be enabled (by following a certain sequence of internal actions) yet the action cannot be guaranteed to be executed through the use of strong fairness or progression (see **system4**, below). In such cases, we require the notion of *possible fairness*. (Lamport has also considered the notion of possibility[23] but our approach, we believe, is much simpler with respect to the development of *fair objects*.)

Definition 5. May — *System Π may lead to P*

$$[[\mathbf{May}_{\Pi}(P)]](\sigma) \triangleq [[\Pi]](\sigma) \Rightarrow \exists \rho \sqsubset \sigma : \exists \tau : [[\Pi]](\rho \cdot \tau) \wedge [[\Diamond P]](\tau)$$

Definition 6. Always possible — *P is always possible in Π*

$$\mathbf{P}_{\Pi}(P) \triangleq [[\Pi]](\sigma) \Rightarrow \forall \rho \sqsubset \sigma : \exists \tau : [[\Pi]](\rho \cdot \tau) \wedge [[\Diamond P]](\tau)$$

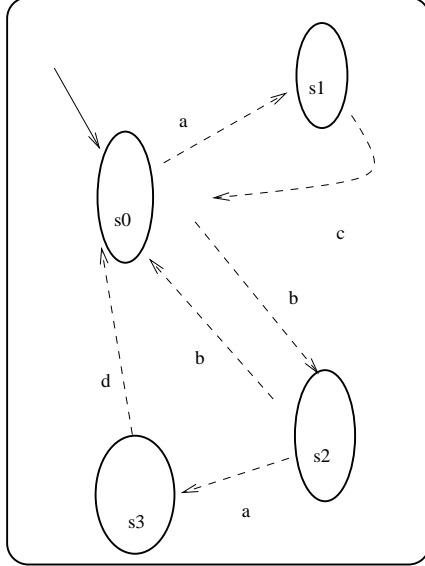
Definition 7. Possible fairness —

An action \mathcal{A} is *possible-fair*, written $PF(\mathcal{A})$, if it is guaranteed to be enabled when it is always possible.

$$PF_{\Pi}(\mathcal{A}) \triangleq \mathbf{P}_{\Pi}(P) \Rightarrow \Diamond Enabled \mathcal{A}$$

A **system** is said to be possible-fair if all its internal actions are possible-fair, written $\text{PF}(\text{system})$. Consider the O-LSTS for **System4**. We see that event **c** is always possible, i.e. no matter what state we are in we can always find a sequence of actions which will lead to **c** being enabled. In such a situation we may wish to specify $\Box\Diamond\mathcal{C}$. Specifying strong fairness does not, however, disallow the trace $\mathbf{b};\mathbf{a};\mathbf{d};\mathbf{b};\mathbf{a};\mathbf{d};\dots$. To disallow this behaviour we require a possible fairness on event **c**.

System4



-----module *System4*-----

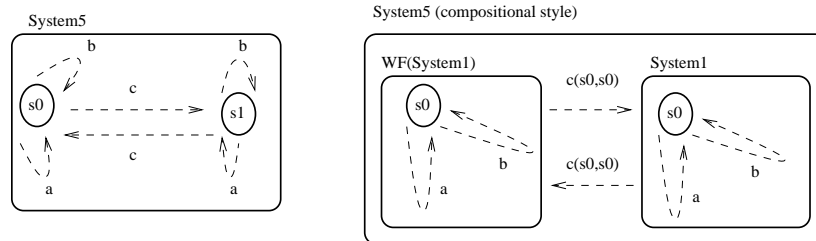
extends DATA
 x, s : VARIABLE
definitions
 STATES $\triangleq \{ s0, s1, s2, s3 \}$
 Initial_System4 $\triangleq (s = s0) \wedge (x \in \text{DATA})$
 $\mathcal{A} \triangleq$
 $\vee R_a(x, x') \wedge (s = s0) \wedge (s' = s1)$
 $\vee R_a(x, x') \wedge (s = s2) \wedge (s' = s3)$
 $\mathcal{B} \triangleq$
 $\vee R_b(x, x') \wedge (s = s0) \wedge (s' = s2)$
 $\vee R_b(x, x') \wedge (s = s2) \wedge (s' = s0)$
 $\mathcal{C} \triangleq$
 $\vee R_c(x, x') \wedge ((s = s1) \wedge (s' = s0))$
 $\mathcal{D} \triangleq$
 $R_d(x, x') \wedge ((s = s3) \wedge (s' = s0))$
 System4 $\triangleq \wedge \text{Initial_System4}$
 $\wedge \Box[\mathcal{A} \vee \mathcal{B} \vee \mathcal{C} \vee \mathcal{D}]_x$
 $\wedge SF \langle x, s \rangle \mathcal{A}$
 $\wedge SF \langle x, s \rangle \mathcal{B}$
 $\wedge PF \langle x, s \rangle \mathcal{C}$
 $\wedge PF \langle x, s \rangle \mathcal{D}$
assumptions
 Assumption_ $\mathcal{A} \triangleq$
 $\forall x \in \text{DATA}, \exists x' \in \text{DATA} : R_a(x, x')$
 Assumption_ $\mathcal{B} \triangleq$
 $\forall x \in \text{DATA}, \exists x' \in \text{DATA} : R_b(x, x')$
 Assumption_ $\mathcal{E} \triangleq$
 $\forall x \in \text{DATA}, \exists x' \in \text{DATA} : R_c(x, x')$

We note that strong fairness on event **a** implies $\Box((\Diamond c) \vee (\Diamond d))$. An **a** event must always eventually be taken (due to the strong fairness condition) and so this **a** event enables either event **c** or event **d**. **System4** is thus modelling a deferred nondeterministic choice between actions **c** and **d**. It is possible to model the same behaviour by renaming all the **a** events, and specifying strong fairness on each of them. This approach would give rise to many complications if the **a** events corresponded to synchronisations between different component objects. Possible fairness lets us reason about such complex cases in a very simple manner.

3.3 Compositional Fairness

There are many different ways in which we may wish to define new *fair objects* in terms of already specified *fair objects*. One such composition mechanism is illustrated by **System5**, where we require the following behaviour with respect to fairness:

Event **c** should act like a fairness switch (which is *on* in state **s0** and *off* in state **s1**). When *on*, events **a** and **b** should be weakly fair. When *off*, there should be no fairness on these events.



Thus, we allow the traces $a;c;b;b;b;\dots$ and $a;b;b;c;a;a;a;\dots$, but disallow an infinite sequence of **a**'s or **b**'s when there has been an even number of **c** events. We cannot specify this behaviour in a satisfactory way by using the global fairness operators. (We can specify the behaviour in a contrived fashion by relabelling events and hiding the specification behind an interface which decodes the relabelling.) Rather, we require a means of specifying **System5** as a composition of two different types (with respect to the fairness) of **System1**. In this way we localise the fairness to system components rather than forcing fairness upon the whole system. This simplifies the proving of theorems using TLA and aids implementation.

3.4 Politeness and Eventuality

We now examine the composition of *fair objects* in such a way that nondeterminism must be resolved by co-operation. First, we need some new definitions:

Definition 8. Eventuality

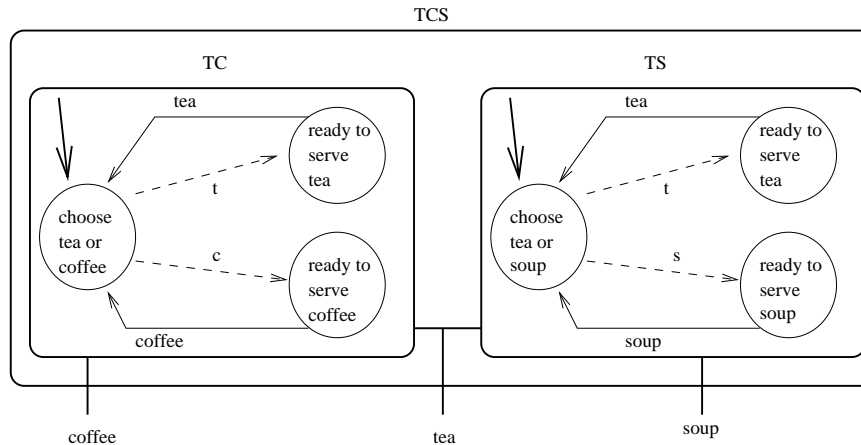
Action **a** is said to be *eventual* in Process **P**, written $ev_a(P)$, iff $P \Rightarrow \Box \Diamond Enabled(a)$

Definition 9. Eventuality set

The *eventuality set* of Process **P**, written $Ev(P)$, is defined to be: $\{x : ev_x(P)\}$

We say that one system is *polite* to another, with respect to some construction operator, if the *eventuality* properties of the events in the first system are maintained in the composed system. This is illustrated by the **TCS** example, below.

TC is a nondeterministic tea-coffee machine. **TS** is a nondeterministic tea-soup machine. The machine **TCS** is a composition of the two which requires synchronisation on the external **tea** action. The example is trivial but it does illustrate the *politeness* concept.



In system **TCS** we say that the components are *impolite* because either one can cause deadlock in the other. Consider a client of **TCS** who requires **tea**, and is willing to wait (a finite period of time) while others are served **coffee** or **soup**. Now, if either machine reaches the **ready to serve tea** state then that component cannot serve another drink until the other is also **ready to serve tea**. Thus, the client may never get their **tea** request served. However, if we specify **SF(TC)** then the tea-coffee machine cannot forever choose to serve coffee and so cannot deadlock the tea-soup machine. In other words **SF(TC)** is polite to **TS** when they synchronise on **tea**. Similarly, **SF(TS)** ensures that the tea-soup machine cannot cause deadlock in the tea-coffee machine.

To formalise this notion, we say that synchronised machines are polite (in a certain composition) if the eventuality sets of each machine is a subset of the eventuality set of the complete system in which they are composed.

3.5 Eventuality Protocols

Consider a queue object which allows the pushing and popping of *elements*; there is one particular exception case which we must consider: what happens when the queue is empty and a pop is requested. Is it the responsibility of the server or the client to cope with such an instance?

In the first instance we consider *client responsibility*. Here, the server queue cannot enable the pop request and thus refuses to participate in it. The client must then either wait until some other client pushes an element onto the queue or decide to do something else (or both). However, the client is unable to distinguish between a queue that is empty and not serving the request, and a queue that is taking its time to service the request. This approach is fundamentally flawed.

In the second instance, we consider *server responsibility*. Here, the notion of politeness is fundamental. If the queue server can always guarantee the enabledness of the pop action (even when the queue is empty) then the client has no additional worries. Within our fair objects framework we have a wide range of means of achieving responsible servers.

Client Eventuality Requirements — We have identified five different types of *client eventuality requirements* which could provide high level reusable concepts. A

client may require that a service request be serviced **immediately**. A client may require that a service is carried out **eventually**. A client may wish a service to be performed **immediately on condition** that if it cannot be done without delay then it will be informed. A client may wish a service to be performed **eventually on condition** that if it cannot be guaranteed to be done in a finite period of time then it will be informed. The client wants the service but places **no eventuality** requirements on when the service must be performed (if ever).

Server Eventuality Properties — Each service that a server offers can be classified dynamically, during execution. An **immediate** service is now enabled. An **eventual** service is guaranteed to be enabled in a finite period of time. An **possible** service may be enabled but it depends on the environment of the server in forcing certain state transitions; no internal action can make the service impossible to fulfil eventually. A **probable** service can possibly be enabled but it depends on some internal nondeterminism; no external service request can make it impossible to service the request (eventually). An **impossible** service will never be enabled.

Client-Server eventuality protocols — Within a fair object system, the *server* properties can, we hope, be made to match the *client* requirements. This is the job of an *interface protocol* which separates clients from servers. In a formal model of requirements we should be able to prove that eventuality needs are fulfilled by servers. For now, we concentrate on how such needs can be specified and how eventuality properties can be guaranteed. The *fairness* classifications which we identified earlier in this paper may be used to provide server guarantees to clients.

4 Telephone Feature Examples

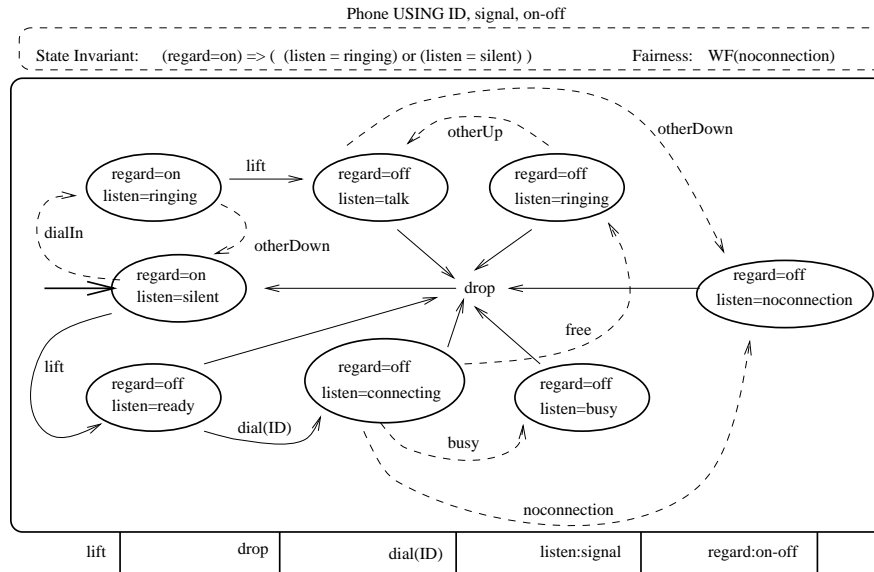
In each of the following examples, we specify the behaviour requirements from the telephone user's point of view. All network operations are modelled as internal nondeterministic operations.

4.1 The Phone

The O-LSTS specification of a telephone is given below.

State invariant properties define restrictions on the possible sets of component values, using the \square operator of TLA. For example, as is shown in the figure below, we may require that when **onhook** the **Phone** *must* be **ringing** or **silent**. These properties are verified, for more complex cases, by proving that all transitions are closed with respect to the invariant, and the invariant is true for initial state(s). Note that the state invariants specified in this way are explicit requirements of the client that must be respected by the model. A specification where the invariants are not true is said to be *inconsistent*.

Liveness conditions can be specified on the nondeterministic events in the model, using the \diamond operator of TLA. For example, we require that when **off** and **connecting** the user does not wait forever for a state transition if they refuse to drop the phone. This must be specified in a separate TLA (temporal) clause. We specify *weak fairness* on the **noconnection** action. Note that we cannot specify the **Phone** to be a weakly fair object since we cannot expect someone to phone us (even if we would like them to).



4.2 An answering machine

Without giving the actual specification, it is clear that an answering machine requires fairness semantics. The standard functionality is for the phone to ring for a finite period of time and then a message to be taken.

An *eventuality* requirement is that when I ring someone with an answering machine I will eventually talk with them or get to leave a message. This requirement can be proven with TLP, using a fair object model of the answering machine. However, as we shall see below, when combined with other features, such an eventuality requirement can be lost (and we have an interaction).

4.3 Call forwarding

Informally, call forwarding can be used to transfer an incoming call to another line. Thus, if I am not at home I can, for example, forward my calls to my portable phone. Here the transfer is an internal action and it must be completed in a finite period of time, and so we once again we have an eventuality requirement. In the specification, we make the requirement conditional on the forwarding number being connectable.

4.4 Answering machine with Call forwarding

Combining these two features illustrates the difference between a fair object model and a standard object model. Without fairness, when I telephone someone with an answering machine who has forwarded their call to another number then, if there is no answering machine at the second phone, I cannot leave a message if they do not reply. With the fair object requirements model, I must be able to leave a message if the person doesn't reply (independent of whether the call is forwarded or not). Thus, if the telephone at the forwarded address does not have an answering machine my specification requires that the call control is returned to the original phone so that a message can be left there.

5 Conclusion

We have shown how the integration of object oriented semantics and temporal logic is appealing. Different types of *fair object* have been introduced and the means of re-using these high-level concepts has been shown. The formalisation of a specification language based on *fair object* concepts is incomplete and we have been forced to specify our problems using the two different semantics. Such a dual-model approach has proven itself in the domain of telecom feature development. We believe that our research could also be useful in any problem domain where clients and servers act in a distributed framework. Certainly, our work is not yet complete, but we are already finding our *fair objects* to be useful conceptualisations during different stages of formal development.

References

1. Lamport, L.: The Temporal Logic of Actions. ACM transactions on programming languages and systems **16** (May 1994) 872 – 923
2. Lamport, L.: The Syntax and Semantics of TLA+ Part 1: Definitions and Modules. Technical Report (Web Pages) (1996)
3. Gibson, J.P.: Formal Object Oriented Development of Software Systems Using LOTOS. PhD Thesis CSM-114, Stirling University (1993)
4. Gibson, J.P.: Formal Object based Design in LOTOS. Stirling University Computing Science TR-113 (1994)
5. Zave, P.: Feature Interactions and Formal Specifications in Telecommunications. IEEE Computer Magazine **aug93** 18–23
6. Bouma, L.G. and Velthuisen, H.: Feature Interactions In Telecommunications. IOS Press (1994)
7. Cheng, K. E. and Ohta, T.: Feature Interactions In Telecommunications III. IOS Press (1995)
8. Gibson, J. P.: Feature Requirements Models: Understanding Interactions. Feature Interactions In Telecommunications IV 46–60 IOS Press.
9. Gibson, J. P., Mermet B. and Méry D.: Feature Interactions: A Mixed Semantic Model Approach. Irish Workshop on Formal Methods, Dublin (1997).
10. Constantine, L.: Beyond the madness of methods: System structure methods and converging design. Software Development. Miller-Freeman (1989)
11. Cutts, G.: Structured system analysis and design method. Blackwell Scientific Publishers (1991).
12. DeMarco, T. Structured analysis and system specification. Prentice-Hall (1979).
13. Booch, G.: Object oriented design with applications. Benjamin Cummings (1991).
14. Coad, P. and Yourdon, E.: Object oriented design. Prentice-Hall (1990).
15. Cox, B.: Object oriented programming: an evolutionary approach. Addison-Wesley (1986).
16. Liskov, B. and Zilles S.: Programming with Abstract Data Types. ACM SIGPLAN Notices **9(4)** (1974) 50–59.
17. Bolognesi, T. and Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems **14** (1987) 25–59.
18. van Eijk, Vissers and Diaz: The Formal Description Technique LOTOS. North-Holland (1989).
19. ISO/IEC: Working Document on Topic 6.2 - Formalisms and Specifications. Information Retrieval, Transfer and Management for OSI **ISO/IEC-JTC1/SC21/WG7** (1989)
20. ISO/IEC: Specification Styles for Structuring of OSI Formal Descriptions **ISO/IEC JTC1/SC21/N669** (1989)
21. Alpern, B. and F. B. Schneider, F. B.: Defining liveness. Information Processing Letters **21(4)** (1985) 180–185.
22. Engberg, U.: TLP Manual-(release 2. 5a)-PRELIMINARY. Department of Computer Science, Aarhus University (1994).
23. Lamport, L.: Proving Possibility Properties. DEC technical report (1995).