Reuse of Invariants in Proofs of Implementation

Anders Gammelgaard

Department of Computer Science, Århus University, Ny Munkegade, DK-8000 Århus C, Denmark

July 1991

Abstract

In this paper we describe a technique to inherit safety properties from abstract programs to their implementations. With this technique repetition of many proofs can be avoided.

Let P be a concurrent program and P' its implementation. The basic idea is taken from [9]: establish a map α from the state space of P' to the state space of P, and map all reachable atomic transitions (s', t') of P' to pairs of states, $(\alpha(s'), \alpha(t'))$, in the state space of P; if each pair can be demonstrated to be either an atomic transition of P or the empty transition, $\alpha(s') = \alpha(t')$, then any safety property of P invariant to arbitrary repetition of states induce a similar safety property of P'.

For many programs some of the unreachable transitions in P' do not map this way; hence it may be necessary to characterise the reachable states of P' in advance of the above demonstration. We prove that the properties we want to inherit can safely be used for this. The main purpose of the paper is to demonstrate the utility of such a result.

The theoretical results in the paper are worked out into gradually more powerful techniques for inheriting safety properties. These proof techniques are illustrated with examples.

1 Introduction

Refining a concurrent shared variable program is difficult. The properties of such a program may vanish when more interleavings are introduced in the process of refinement, And this may happen even by splitting a single atomic action into a few actions. Thus a refmement to part of the program can affect the whole program; in general it must be treated as a refinement to all of the program.

As consequence it is difficult to adopt well-known techniques from sequential programming for making structured development of programs along with their correctness proofs. In lack of such techniques, the whole correctness proof is often repeated in each step of the refinement process; at best a former proof is used as a guide how to make the next proof in the refinement process. For two examples of this, see [5] and [7].

In this paper we describe a technique to avoid the repetition of proofs of safety properties. A traditional proof is replaced by a set of relatively simple checks which utilise the already proven properties.

The basis of the technique is an observation made by Lamport in [9] and worked out in [8], [10], and [1]. Let P be a program and let P' be its implementation. Usually state transitions in P will split into more transitions in P'. For instance a transition from state s_0 to state s_1 in P may be implemented by the sequence of state transitions $s'_0, \ldots, s'_n$. Now, if repetition of states is ignored, executions of P' may be considered as executions of P – we just have to map states of P' to states of P such that exactly one of the n state transitions (s'_i, s'_{i+1}) map to the state transition (s_0, s_1); the other transitions just map to repetitions of states.

Some states in P' may not be reachable, however. Transitions from such states need not and usually do not map to P-transitions or empty transitions. To restrict attention to only reachable transitions we need a characterisation of the reachable states in P'. This can be attained by proving invariants in P'. But sometimes such invariants are not orthogonal with the properties we want to inherit. They may even coincide with inherited invariants. Then the basic technique does not seem to be useful: Before the inheritance the invariant must anyway be proven true in P' by traditional means.

In this paper we prove two theorems to cope with such cases. We show that it *is* possible to use inherited properties as characterisations of the reachable states. Furthermore a property of P' may be used as characterisation if it can be proven *relatively* invariant, i. e. invariant under the assumption that inherited properties hold in P'.

The basic simulation idea is taken from Lamport ([9], [8], [1], [10]). In his terminology the map from states of P' to states of P is called a refinement mapping.

Similar ideas have also been put forward by other authors. For instance Lynch and Tuttle [11] prove a low level description of an arbiter correct by treating it as an implementation of a high level description. In their approach two levels must be related by a so-called possibilities mapping which corresponds to a refinement mapping except that each low level state is mapped to a set of high level states instead of just a single high level state. In order to prove that a map h is a possibilities mapping one must roughly show that for each $s_0 \in h(s'_0)$ such that s'_0 is a reachable low level state and s_0 is a reachable high level state if (s'_0, s'_1) is a low level transition, then either $s_0 \in h(s'_1)$ or there is an s_1 such that $s_1 \in h(s'_1)$ and (s_0, s_1) is a high level transition. Because of the restriction to reachable high level states they are able to reuse already proven high level invariants much as in the present paper. But because both the high and the low level reachable states must be found independently, they are not able to use high level invariants in proofs of the relative invariance of low level properties. Furthermore Lynch and Tuttle do not present the possibility of reusing high level properties as an ability to inherit properties in advance of the proof.

Another approach is presented by Jonsson in [6]. Instead of a function mapping low level states to (sets of) high level states Jonsson requires that a relation R between the two levels be defined. The obligation then roughly is to show that if relation $s'_0 R s_0$ holds and (s'_0, π, s'_1) is a low level transition with label π, then there exists a high level state s_1 and a high level transition (s_0, π, s_1) such that $s'_1 R s_1$ holds. (If pi is an "invisible" label, then the high level transition could just be repeating the state since such repetition is always allowed by Jonssons machine descriptions.) When using functions mapping from low level states to (sets of) high level states the low level reachable states must be characterised by invariants. When using relations these invariants can instead be expressed by restricting the domain of the

relations. This means that in Jonssons implementation proof one also proves low level invariants: For each pair $s_0' R s_0$ and transition (s_0', π, s_1') one of the requirements is to find an s_1 such that relation $s_1' R s_1$ holds, that is one must show that s_1' belongs to the domain of R. The reuse of already proven high level invariants seems not to be present in Jonssons work. But it is probably just a small change to allow the same amount of reuse as in Lynch and Tuttle's work.

The structure of the paper is as follows. In section 2 some basic concepts and some notation is introduced. Section 3 is devoted to the basic idea of inheriting safety properties. In section 4 we go on to the main result of the paper. We prove two theorems which show that it is possible, in implementation proofs, to use inherited properties to derive characterisations of the reachable low level states. The results are summed up into an inheritance technique. In section 5 we illustrate with examples how to use the technique The aim is a work out of the technique which is both simple and widely applicable. In section 6 a further extension of the technique is presented. It is more powerful but may also be more difficult to master technically. Section 7 contains a discussion of the completeness of the technique presented in section 6. Finally, in section 8, we discuss the applicability of the technique and we discuss the possibility of further extensions to the technique.

2 Sequences, properties, and program executions.

We introduce notation to describe sequences and sets of sequences.

In the following let $\Sigma = \{s, t, \ldots\}$ be some set of states. Σ^+ denotes the set of finite non-empty – Σ^ω the set of infinite sequences over Σ. For $\phi \in \Sigma^+$ and $\psi \in \Sigma^+ \cup \Sigma^\omega$ the concatenation of ϕ and ψ is written $\phi * \psi$; the last state of ϕ is denoted by $\phi^\bullet$; hence $(\phi * s)^\bullet = s$ holds for any state $s \in \Sigma$ (we treat Σ as a set of single element sequences). The natural prefix ordering between sequences is denoted by the symbol $\sqsubseteq$.

Any subset $\Gamma \subseteq \Sigma^+ \cup \Sigma^\omega$ is called a *property*; sequences Γ are said to *have property* Γ. Γ is *prefix closed* if $\psi \in \Gamma$ and $\phi \sqsubseteq \psi$ implies $\phi \in \Gamma$ for non-empty ϕ; and Γ is *ω-complete* if the least upper bound of any chain

$\phi_1 \sqsubseteq \phi_2 \sqsubseteq \ldots$ is an element in Γ provided each approximation ϕ_i is in Γ.

In this paper we study a class of properties called safety properties. Following [2] and [3] a safety property stipulates that no bad thing ever happens. It is implicit that the bad thing is some event occurring within finite time: if no bad thing ever happens in any prefix of a sequence, then the entire sequence is accepted. So safety properties are ω-complete.

We furthermore have ([2] [3]) that an occurrence of a bad thing is irremediable; any extension of a bad sequeuce remains bad. In other words if a sequence has some safety property, then all its prefixes also have this property. So safety properties are prefix closed.

> $\Gamma \subseteq \Sigma^+ \cup \Sigma^\omega$ is a *safety property* iff Γ is ω-complete and prefix closed.

All properties treated in the examples of this paper are safety properties. This is easily verified in each case.

Programs are modelled by transition systems. A *transition system* $P = (\Sigma, \rhd, A)$ consists of a set of states Σ, a transition relation $\rhd \subseteq \Sigma \times \Sigma$, and a set of initial states $A \subseteq \Sigma$. For P we define the set of *finite P-executions* $[A]\rhd$ to be the least set satisfying

1. $A \subseteq [A]\rhd$ and
2. if $\phi \in [A]\rhd$ and $(\phi^\bullet, s) \in \rhd$, then $\phi * s \in [A]\rhd$.

The least ω-complete set containing $[A]\rhd$ is called the *P-executions* and is denoted $\overline{[A]\rhd}$. Because any safety property Γ is ω-completes we immediately get that to establish that all executions in $\overline{[A]\rhd}$ have property Γ it is sufficient to establish $[A]\rhd \subseteq \Gamma$.

3 Basic technique

Let $P = (\Sigma, \rhd, A)$ and $P' = (\Sigma', \rhd', A')$ be two transitions systems. P' should be regarded as an implementation of the abstract program P. Entities from P' are attached with a prime to distinguish them from P-entities.

A simple way of relating P' to P is to choose a map α from Σ' to Σ. Element-wise application of α induce similar maps from pairs of Σ'-states to pairs of Σ-states and from sequences over Σ' to sequences over Σ. Without ambiguity all these maps are denoted by α.

$[A']\rhd'$ can be related to $[A]\rhd$ if α maps A' to A and $\rhd'$ to $\rhd$. It is not necessary to consider transitions in $\rhd'$ from unreachable states. The *reachable states* R in transition system P are the states occurring in $[A]\rhd$. For any $M \subseteq \Sigma$ we define ${}_M|\rhd$, the $\rhd$-transitions *from* M, to be the set $\{(s,t) \in \rhd : s \in M\}$; then the *reachable transitions* in S are the elements in ${}_R|\rhd$.

> **Inheritance theorem.** *Let α be a map from Σ' to Σ. If $\alpha(A') \subseteq A$ and $\alpha({}_{R'}|\rhd) \subseteq \rhd$, then $\alpha([A']\rhd) \subseteq [A]\rhd$.*
>
> *Proof.* We prove that α maps $[A']\rhd'$ to $[A]\rhd$ by induction on the length of sequences in $[A']\rhd'$.
> Basis: A' is the set of one-element sequences in $[A']\rhd'$; so $\alpha(A') \subseteq A$ and $A \subseteq [A]\rhd$ gives the conclusion.
> Induction step: Let $\phi * s$ be a $(k+1)$-element sequence in $[A']\rhd'$ and assume that $\alpha(\phi) \in [A]\rhd$. Since $\phi * s$ is in $[A']\rhd'$, the pair $(\phi^\bullet, s)$ must be a transition in ${}_{R'}|\rhd'$; so $\alpha(\phi^\bullet, s)$ is in $\rhd$. Then $\alpha(\phi) * \alpha(s)$ and hence also $\alpha(\phi * s)$ is in $[A]\rhd$.

From the theorem we immediately conclude

> **Inheritance corollary 1**. *Let the finite P-executions have property Γ (that is $[A]\rhd \subseteq \Gamma$), and let α be a map from Σ' to Σ. If $\alpha(A') \subseteq A$ and $\alpha({}_{R'}|\rhd') \subseteq \rhd$, then $\alpha([A']\rhd') \subseteq \Gamma$ – i. e. the finite P'-executions have property $\alpha^{-1}(\Gamma)$.*

The corollary shows that P-properties can be used to induce new P'-properties or, as we shall also say, P-properties can be inherited. A P-property and its corresponding P'-formulation are often very similar. One such similarity is expressed in the following proposition. It is important for the applicability of the whole technique and is easily seen to be true.

> **Proposition.** *If Γ is a safety property, then also $\alpha^{-1}(\Gamma)$ is.*

Property Γ is said to be *invariant to repetition of states* if it satisfies that $\phi * \psi \in \Gamma$ holds if and only if $\phi * \phi^\bullet * \psi \in \Gamma$ holds. Such properties are useful in hierarchical construction of programs [9]: Any action considered atomic at one level may split into a set of actions at a level below causing lower level executions to contain more state transitions than corresponding higher level executions. The best we can hope for is to construct a map from lower level states to higher level states such that just one of the transitions map to a transition caused by the abstract action; the other transitions just map to repetitions of the current state.

Inheritance corollary 1 cannot directly utilise invariance to repetition of states. But observe that if executions of $(\Sigma, \rhd, A)$ have property Γ where Γ is invariant to repetition of states, then also executions of $(\Sigma, \trianglerighteq, A)$ have this property where $\trianglerighteq$ is the reflexive closure of $\rhd$ – i. e. $\trianglerighteq = \{(s, s) : s \in \Sigma\} \cup \rhd$. Applied to inheritance corollary 1 this observation allows us to relax the requirement $\alpha({}_{R'}| \rhd') \subseteq \rhd$:

> **Inheritance corollary 2.** *Let the finite executions of $P = (\Sigma, \rhd, A)$ have a property Γ invariant to repetition of states, and let α be a map from Σ' to Σ. If $\alpha(A') \subseteq A$ and $\alpha({}_{R'}|\rhd') \subseteq \trianglerighteq$, then executions in $[A']\rhd'$ have property $\alpha^{-1}(\Gamma)$.*

Before we go on to show how to use the technique we develop some notation for mappings between state spaces.

In most applications Σ will be the Cartesian product of the value spaces of some program variables $v_1, \ldots, v_n$ of P. We treat each variable v_i as a projection $v_i : \Sigma \to [v_i]$, where $[v_i]$ is the value space of v_i. Hence s is determined by the values $v_1(s), \ldots, v_n(s)$. Furthermore any function $\alpha : \Sigma' \to \Sigma$ is determined by the n equations

$$v_i \circ \alpha \equiv e_i, \quad \text{for } i = 1, \ldots, n$$

Here e_i is a function from Σ' to $[v_i]$.

The characteristic function F_Γ of a property Γ is the boolean valued function on $\Sigma^+ \cup \Sigma^\omega$ satisfying $\phi \in \Gamma$ if and only if $F_\Gamma(\phi)$ is true. It is easy to inherit Γ given F_Γ; the property $\alpha^{-1}(\Gamma)$ is simply the set of sequences on which $F_\Gamma \circ \alpha$ is true. So the function $F_{\alpha^{-1}(\Gamma)}$ is identical with $F_\Gamma \circ \alpha$.

Now assume that F_Γ is given by an expression in the program variables $v_1, \ldots, v_n$ and that $\alpha : \Sigma' \to \Sigma$ is given by n equations $v_i \circ \alpha \equiv e_i$. Then $F_\Gamma \circ \alpha$ is an expression in $v_1 \circ \alpha, \ldots, v_n \circ \alpha$. By using the n equations this reduces to an expression in $e_1, \ldots, e_n$. Hence $F_\Gamma \circ \alpha$ may be expressed by $F_\Gamma[v_1, \ldots, v_n/e_1, \ldots, e_n]$.

Example A. Consider a buffer with maximum capacity $N > 0$ messages shared between a producer process p and a consumer process c. The producer must be prevented from overflowing the buffer while the consumer must not attempt to remove messages from an empty buffer. Deadlock may not occur.

This programming problem is solved by introducing two semaphores [4] a' and b', where a' represents the unused space and b' the space in use;

integer a', b' $(a' = N, b' = 0)$;
cobegin
p : **repeat** $wait_p$: $P(a')$; $signal_p$: $V(b')$ **forever** $\|$
c : **repeat** $wait_c$: $P(b')$; $signal_c$: $V(a')$ **forever**
coend

wait and *signal* are program locations. Processing of messages is not shown but manipulations of buffer space is assumed to take place between the $P(\cdot)$- and $V(\cdot)$-primitives. The program defines a transition system with state space spanned by a', b', and the program counters of p and c. Each program counter has just two values, *wait* and *signal*. The subscripts p and c are added to enable easy reference to a specific program counter along with its current value.

We use predicates to define sets of states. The predicate (at $wait_p$) is satisfied by any state where the value of the producer's program counter is *wait*. The initial states satisfy the predicates $(a' = N)$, $(b' = 0)$, (at $wait_p$), and (at $wait_c$); and the transition relation is defined in accord with the traditions semantics of semaphores; e.g. $P(a')$ describes the set of transitions (s', t') where (at $wait_p$) $\wedge a' > 0$ holds in s' and where a' gets decremented by 1 and (at $signal_p$) holds in t'.

The program is deadlocked if both processes are at *wait*, $a' = 0$, and $b' = 0$. We prove the program deadlock-free by treating it as an implementation P' of an abstract program P. In P the compound actions $P(a')$; $V(b')$ and

$P(b')$; $V(a')$ are both considered atomic. This is indicated by brackets $<$ and $>$.

integer a, b $(a = N, b = 0)$;
cobegin
  $p:$ **repeat** $< P(a); V(b) >$ **forever** $\|$
  $c:$ **repeat** $< P(b); V(a) >$ **forever**
coend

Since each program counter has just a single value, there is no need to include information of program counters in the state space of P. So Σ is spanned by just a and b. Each of the two actions in P can execute only when $a > 0$ respectively $b > 0$.

Let I be a predicate on states and assume $M \subseteq \Sigma$ is the set of states on which I is true. Let Γ be the property $M^+ \cup M^\omega$. Then F_Γ is true on ϕ exactly when I holds for all states in ϕ. Separate notations will not be used to discern I, M, Γ, and F_Γ; the intended meaning can be inferred from the context.

It is trivial to show that the P-executions have property

$$I: \qquad a + b = N$$

This implies that $a > 0 \vee b > 0$ holds and we thus have deadlock freedom in P.

A similar argument for deadlock freedom in P' would use that $a' + b' = N$ holds when the predicates (at $wait_p$) and (at $wait_c$) are both true. But $a' + b' = N$ is not a property of P' as $a' + b'$ can take on any of the values N, $N - 1$ and $N - 2$. A more complicated invariant using all these values must be introduced to do the same job.

Instead we use the inheritance technique to get the property

$$(\text{at } wait_p) \wedge (\text{at } wait_c) \Rightarrow a' + b' = N$$

Let $q_1 \rightarrow e_1 \;\Box\; q_2 \rightarrow e_2 \;\Box \cdots \Box\; e_n$ denote the function which has value e_i, $1 \leq i < n$ if $q_1, \ldots, q_{i-1}$ all evaluate to false and q_i evaluates to true, and which has value e_n if all $q_1, \ldots, q_{n-1}$ evaluate to false. Then α can be defined by

$$a \circ \alpha \equiv (\text{at } wait_p) \rightarrow a' \ \Box \ a' + 1$$
$$b \circ \alpha \equiv (\text{at } wait_c) \rightarrow b' \ \Box \ b' + 1$$

Hence α suppresses the state changes caused by $P(\cdot)$-primitives.

We now use this definition when both processes are at *wait* in some state s'; if $\alpha(s')$ satisfies I then s' satisfies $a' + b' = N$; this is the desired implication.

To establish the property $I \circ \alpha$ we use inheritance corollary 2 with I as the property Γ.

Clearly $\alpha(A') \subseteq A$.

To prove $\alpha(_{R'}| \rhd') \subseteq \trianglerighteq$ we need a characterisation of R'. The P-primitives prevent a' and b' from becoming negative. We will use this fact to prove that e. g. the $V(b')$ action in P' maps to the entire $< P(a); V(b) >$-action in P; if for instance a' were -1 at $signal_p$ the execution of $V(b')$ would map to a state change where $< P(a); V(b) >$ executes in spite of the fact that a has value 0. So we are looking for a characterisation $I' \supseteq R'$ for which we *can* prove that $\alpha(_{I'}|\rhd') \subseteq \trianglerighteq$. A sufficient characterisation is the following property which is easily proven to hold for P'-executions.

$$I': \quad ((\text{at } signal_p) \Rightarrow a' \geq 0) \wedge ((\text{at } signal_c) \Rightarrow b' \geq 0),$$

The relation $\alpha(_{R'}|\rhd') \subseteq \trianglerighteq$ is proven by means of a schema. Such schemas will be used repeatedly in this paper; hence we give it a detailed description here.

action	cases	I'	maps to	final state	maps to	image
$wait_p$	$a' = n$ [1] $[n > 0]$	[2]	$a = n$ [3]	$a' = n - 1$ [4] $signal_p$	$a = n$ [5]	EMPTY [6]
$signal_p$	$a' = n$ [7] $b' = m$	$[n \geq 0]$ [8]	$a = n + 1$ [9] $b = \hat{m}$	$a' = n$ [10] $b' = m + 1$ $wait_p$	$a = n$ [11] $b = \hat{m} + 1$	[12] $< P(a); V(b) >$

Each row of the schema proves $\alpha(\rhd') \subseteq \trianglerighteq$ for the subset of $_{I'}|\rhd'$ defined by a single atomic action of the program.

The double lines separate each row into 4 parts. The first part defines the considered action by giving its program location The second part treats the left state of those transitions (s', t') defined by the action. Under "cases" it is possible to split the analysis into different cases (not necessary in this example), values of program variables can be specified by parameters (such

as m and n), and semantic restrictions on the execution can be given (such as $a' > 0$ before execution of $P(a')$); I' gives further restrictions obtained from characterisations of R'; and "maps to" describes the image state $\alpha(s')$. The third part treats t', "final state" t' itself and "maps to" its image $\alpha(t')$. Finally the fourth part sum up the results in the two columns labelled "maps to" and points out an action defining the transition $(\alpha(s'), \alpha(t'))$; EMPTY indicates that $\alpha(s') = \alpha(t')$; for $\alpha(s') \neq \alpha(t')$ either the P-action or its program location is given.

Note that we use abbreviations. All predicates in a single entry are to be and'ed together; furthermore predicates for program locations are defined by just writing the desired location. Hence the entry labelled 4 is an abbreviation for the formula $(a' = n - 1) \wedge (\text{at } signal_p)$.

To simplify the schema a number of conventions apply to each row

- P'-variables not mentioned under "final state" retain their value in the transition (s', t').
- $P-$variables not mentioned in the "maps to"-column following "final state" retain their value in the state change $\alpha(s', t')$.
- Parameters have a value which is fixed for the entire row. Conditions on parameters are enclosed in angle brackets and apply to the entire row.
- Numbers may be attached to the entries in order to explain how they are obtained.

The schema is only given for the two producer actions since the arguments are symmetric for the two processes. To illustrate all aspects explanations are given for every entry of the schema.

1. a' has some value n in the initial state s' and, since the action at $wait_p$ can execute, this value is positive.
2. No restrictions are needed in this row to prove $\alpha(s', t') \in \trianglerighteq$.
3. Inspection of the α-definition shows that $a = n$ holds in $\alpha(s')$.

4. a' gets decremented by 1 and the control changes to $signal_p$ in the transition.

5. By inspection of the α-definition.

6. No variable has changed its value in the transition (a has not since $a = n$ holds in both states, b because it is not mentioned). Hence $\alpha(s', t')$ is just a repetition of the state.

7. a' and b' have some values before the execution.

8. The invariant I' gives that a' is positive.

9. By inspection of the α-definition $a = n + 1$ holds in $\alpha(s')$ and b has some value $\hat{m}$ (which is one of m or $m + 1$).

10. b' is incremented by 1 and control changes to $wait_p$.

11. By inspection of the α-definition $a = n$ holds in $\alpha(t')$; since the program counter of the consumer has not changed, the same case of the definition as in 9 must be used for the b-value; and as b is incremented, we have $b = \hat{m} + 1$.

12. From 8 and 9 a is positive in $\alpha(s')$; b gets incremented and a decremented by one. Hence $\alpha(s', t')$ corresponds to a transition defined by the producer action $< P(a); V(b) >$.

We have proved that P' cannot deadlock; i. e. a state where both processes are at *wait* and $a' = b' = 0$ cannot occur.

To prove liveness properties we may wish to know more safety properties of P': Even when $a' + b' = 1$ holds forever, we cannot be sure that one of $P(a')$ or $P(b')$ eventually gets executed; a' and b' might alternately have the value 1, and if we only require that $P(v)$ gets executed eventually if $v > 0$ holds continuously, then freedom from deadlock is not sufficient; we need the stronger assertion that $a' > 0$ ($b' > 0$) holds as long as the producer (consumer) is at *wait*. But we now show how to inherit this property also – without additional proof obligations.

If $a > 0$ in program P, then this clearly holds as long as $< P(a); V(b) >$ is not executed. In P' this gives that as long as $V(b')$ is not executed, $a > 0$

holds in $\alpha(s')$; in particular, if the producer remains at *wait*, a' equals $a \circ \alpha$, so a remains positive.

This shows that inherited safety properties may also be useful in the proof of liveness properties.

4 Extensions

In some cases the characterisation of R' by giving a condition that is weaker that R' seems to require too much work. If we want to inherit I and the needed characterisation is identical with or can be deduced from $\alpha^{-1}(I)$, then the technique seemingly does not offer any help as we must anyhow give a traditional proof that P'-executions have property $\alpha^{-1}(I)$.

Example B (constructed for the purpose of illustration). Let P be the program

integer a, x_1, x_2 $(a = 1)$;
cobegin
    repeat $wait_1$: $P(a)$; inc_1: $x_1 := x_1 + 1$; $signal_1 : V(a)$ **forever** $\|$
    repeat $wait_2$: $P(a)$; inc_2: $x_2 := x_2 + 1$; $signal_2 : V(a)$ **forever**
coend

In this program x_1 and x_2 are, in arbitrary order, repeatedly incremented by 1. The increment is performed in two mutually exclusive regions each having program locations *inc* and *signal*.

Now consider an implementation P' where the increment is done by means of a shared temporary variable t.

integer repeat a, x_1', x_2', t' $(a = 1)$;
cobegin
    repeat $wait_1'$: $P(a')$; $read_1''$: $t' := x_1'$; inc_1': $x_1' := t' + 1$;
        $signal_1' : V(a')$ **forever** $\|$
    repeat $wait_2'$: $P(a')$; $read_2' : t' := x_1'$; inc_2: $x_2 := t' + 1$:
        $signal_2' : V(a')$ **forever**
coend

Primes attached to program locations are, as before, just used to ease the reference to a program together with a location. The program counters loc_1' and loc_2' do *not* have primed values; their value spaces are merely extensions of the corresponding loc_1- and loc_2-spaces. The natural definition of the map α is

$$\begin{array}{lcl} loc_i \circ \alpha & \equiv & (\text{at } read_i) \rightarrow inc_i \,\square\, loc_i' \\ a \circ \alpha & \equiv & a' \\ x_i \circ \alpha & \equiv & x_i' \end{array}$$

If we want to inherit a property of P – e. g. that the sequences of x_1- and x_2-values are monotonically non-decreasing – then we must establish $\alpha({}_{R'}|\rhd') \subseteq \trianglerighteq$; but the action $x_i' := t' + 1$ only maps to the corresponding action $x_i := x_i + 1$ provided $\{t' = x_i'\}$ holds before the execution. To prove this relationship the mutual exclusion in P' must be used, and hence also proved. But if such a proof has already been performed in P a simile proof in P' seems to be nothing but a repetition of the proof in P. (End of example.)

Notice that we cannot simply presuppose that we work only with reachable states. It requires some real work to find out what the reachable states are.

The following lemma shows that inherited properties actually *can* be used as characterisations of the reachable states – so to speak in advance of using the inheritance corollaries.

Characterisation lemma 1. *If $R \subseteq I$ (the reachable states of P satisfy I), $\alpha(A') \subseteq A$, and $\alpha({}_{\alpha^{-1}(I)}|\rhd') \subseteq \trianglerighteq$, then $R' \subseteq \alpha^{-1}(I)$.*

Proof. We prove by induction that all sequences ϕ in $[A']\rhd'$ satisfy $\alpha(\phi^\bullet) \in R$. Because $[A']\rhd'$ is prefix closed all states in R' can be written $\phi^\bullet$ for some $\phi \in [A']\rhd'$. So $\alpha(R') \subseteq R \subseteq I$.
Basis: One-element sequences belong to A' which maps to $A \subseteq R$.
Induction step: Let $\phi \in [A']\rhd'$ satisfy $\alpha(\phi^\bullet) \in R$. If $(\phi^\bullet, s') \in \rhd'$ for some s', then $(\phi^\bullet, s')$ also belongs to ${}_{\alpha^{-1}(I)}| \rhd'$ since $R \subseteq I$; so $\alpha(\phi^\bullet, s') \in \trianglerighteq$. If $\alpha(\phi^\bullet) = \phi(s')$ we obviously have $\alpha(s') \in R$. And if $\alpha(\phi^\bullet, s') \in \rhd$, then $\alpha(s')$ belongs to R since R is stable under $\rhd$.

Remark. If the assumptions in the lemma hold, we also have $\alpha({}_{R'}|\rhd') \subseteq$

$\trianglerighteq$; hence the assumptions in inheritance corollary 2 hold, and we can immediately inherit properties from P.

Example C. A subset $I \subseteq \Sigma$ is said to be *invariant* in $P = (\Sigma, \triangleright, A)$ if $A \subseteq I$ and if $s \in I \wedge (s,t) \in \triangleright$ implies $t \in I$ (written $\triangleright(I) \subseteq I$). An ordinary proof by induction shows that the reachable states R is a subset of I when I is invariant. We will demonstrate that this fact may also be concluded by means of the characterisation lemma.

Let $P = (\Sigma, \triangleleft, A)$ be the system where $\Sigma = \{true, false\}$, $\triangleright$ does not contain the transition $(true, false)$, and $A = \{true\}$. Clearly $\{true\}$ is invariant in P. Assume we will prove I' invariant in some other system P'. We define α by

$$\alpha \equiv in\ I' \rightarrow true\ \Box\ false,$$

where $in\ I'(s')$ holds when $s' \in I'$. If $A' \subseteq I'$ holds also $\alpha(A') \subseteq A$ holds; and if $(s' \in I' \wedge (s', t') \in \triangleright')$ implies $t' \in I'$ then all (s', t') with $s' \in \alpha^{-1}(true)$ map to the empty transition – i.e. $\alpha({}_{\alpha^{-1}(true)}|\ \triangleright') \subseteq \trianglerighteq$. The characterisation lemma then gives that $R' \subseteq \alpha^{-1}(true) = I'$ as desired. Hence ordinary proofs of invariants can be considered as just a special case of the technique.
(End of example.)

Example D. Assume a program P is developed using "secure" operations so that division by zero, use of array indices out of range, etc. all cause the program to terminate in an error state. We call states where such a termination may occur in the next transition for dangerous states. If P is demonstrated never to enter dangerous states – either by proof or by exhaustive program test – then any implementation P' can use corresponding "insecure" operations. The informal argument for this may be formalised by the inheritance technique.

The informal argument goes: For all executions not entering dangerous states the P- and P'-operations cause identical state changes. So if P never enters dangerous states, P'-operations will cause identical state changes and may thus be used in place of the P-operations.

Using the inheritance technique we take I to be the undangerous states of P and α to act as the identity function on all undangerous states of P' – hence $\alpha^{-1}(I)$ equals I. The assumption that P'- and P- actions cause identical state changes on undangerous states now gives that $\alpha({}_{I'}|\ \triangleright') \subseteq \triangleright$; so by the characterisation lemma we get $\alpha({}_{R'}|\ \triangleright') \subseteq \triangleright$; and hence by the

remark accompanying the lemma, all safety properties of P may be inherited. (End of example.)

Example B (continued). We now prove $\alpha(_{R'}| \rhd') \subseteq \trianglerighteq$ for the programs P and P' introduced earlier. To inherit that $\{x'_i = t'\}$ holds before execution of $x'_i := t' + 1$ we use a little trick. States not satisfying the desired equality are mapped to states not reachable in P. We could use e.g. states where both processes are between the $P(a)$- and the $V(a)$-operations; but we prefer to introduce an entirely new state $\perp$ into the state space of P acting as *false* in example C. In the thus changed program $P_\perp$ a state only assigns values to variables if the state is different from $\perp$. We note that if P-executions have property Γ, then $P_\perp$-executions also have this property since the introduction of $\perp$ does not change $[A]\rhd$. We now define α by:

$$\begin{array}{lcll} \alpha & \equiv & (\text{at } inc'_i) \wedge (x_i \neq t') \rightarrow \perp \,\Box\, \beta, & \text{where } \beta \text{ is defines by} \\ loc_i \circ \beta & \equiv & (\text{at } read_i) \rightarrow inc_i \,\Box\, loc'_i & \\ a \circ \beta & \equiv & a' & \\ x_i \circ \beta & \equiv & x'_i & \end{array}$$

We will demonstrate that the conditions in characterisation lemma 1 are met; by the accompanying remark this allows us to inherit properties from P. The set I is chosen to consist of those states different from $\perp$ which satisfy $\neg((\text{at } inc_1) \wedge (\text{at } inc_2))$. $P_\perp$-executions are easily seen to have property I.

We immediately get $\alpha(A') \subseteq A$. To demonstrate $\alpha(_{\alpha^{-1}(I)}| \rhd') \subseteq \trianglerighteq$ we use a schema like the one introduced in example A. Now we use a column labelled $\alpha^{-1}(I)$ in place of the column labelled I', however,

action	cases	$\alpha^{-1}(I)$	maps to	final state	maps to	image
$wait'_i$	$a' = n$ $[n > 0]$		$a = n$ $wait_i$	$a' = n - 1$ $read'_i$	$a = n - 1$ inc_i	$wait_i$
$read'_i$	$x'_i = n$		$x_i = n$ inc_i	$t' = n$ inc'_i	$x_i = n$ [1] inc_i	EMPTY
inc'_i	$x'_i = n$ $t' = m$	$[n = m]$ [2]	$x_i = n$ inc_i	$x'_i = n + 1$ [3] $signal'_i$	$x_i = n + 1$ $signal_i$	inc_i
$signal'_i$	$a' = n$		$a = n$ $signal_i$	$a' = n + 1$ $wait'_i$	$a = n + 1$ $wait_i$	$signal_i$

All assertions in the schema are trivially obtained except the following three:

1. Since $t' = n$ function α does not map the state to $\perp$.
2. We have just introduced parameters n, m for the values of x'_i and t'; as $\perp \notin I$ it must be the case that $x'_i = t'$ – i.e. $n = m$.

3. That $x_i' := t'+1$ establishes $x_i' = n+1$ follows from $t' = m$ and $[n = m]$.

Note that mutual exclusion does not enter into the argument at any place. The effect of mutual exclusion is hidden in the inherited assertion that when the control is at inc_i, the variables x_i and t have identical values (End of example.)

Sometimes it may seem awkward to characterise R' by means of the inherited property $\alpha^{-1}(I)$. In the previous example, for instance, t' is a variable only present in the program P'; so to restrict the values of t' by a property inherited from P appears to introduce too much trickery. Intuitively the only property from P needed to establish that (at inc_i') implies $x_i' = t'$ is mutual exclusion in P. Based on mutual exclusion it should be possible to establish $x_i' = t'$ exclusively by arguments in P'. The following generalisation of characterisation lemma 1 shows that it is possible to cover this intuition more accurately.

In the inheritance theorem it could be necessary to characterise R' by an invariant I' proven in P'. This possibility is now relaxed such that I' only needs to be proven relatively invariant: for transitions (s', t') with s' in both I' *and* $\alpha^{-1}(I)$ the relation $t' \in I'$ It must hold.

Characterisation lemma 2. *If* $R \subseteq I$, $\alpha(A') \subseteq A, A' \subseteq I'$, ${}_{\alpha^{-1}(I)}|\rhd'(I') \subseteq I'$ *(relative invariance of* I'*),*
and $\alpha({}_{\alpha^{-1}(I)\cap I'}|\rhd') \subseteq \trianglerighteq$,
then $R' \subseteq \alpha^{-1}(I) \cap I'$.

Proof. Let $\perp$ be a state not occurring in Σ. Let $P_\perp$ be the transition system $(\Sigma \cup \{\perp\}, \rhd, A)$. We define a new function $\beta : \Sigma' \to \Sigma \cup \{\perp\}$ by

$$\beta \equiv in I' \to \alpha \,\Box\, \perp$$

Since $\perp \notin I$, this definition gives $\beta^{-1}(I) = \alpha^{-1}(I) \cap I'$. The idea is to show that β satisfies all conditions in characterisation lemma 1.

As $\alpha(A') \subseteq A$ and $A' \subseteq I'$, also $\beta(A') \subseteq A$ holds. From ${}_{\alpha^{-1}(I)}|\rhd'(I') \subseteq I'$ we get that, if $(s', t') \in \rhd'$ and $s' \in \alpha^{-1}(I) \cap I'$, then

$\beta(t') = \alpha(s')$. So $\alpha({}_{\alpha^{-1}(I)\cap I'}|\ \rhd') \subseteq \trianglerighteq$ gives $\beta({}_{\beta^{-1}(I)}|\ \rhd') \subseteq \trianglerighteq$. Using characterisation lemma 1 now gives $R' \subseteq \beta^{-1}(I)$ as desired.

To obtain a workable technique we sum up the results in characterisation lemma 2, the safety theorem, the inheritance theorem and its corollary 2.

Inheritance technique. *Assume that* $P = (\Sigma, \rhd, A)$ *and* $P' = (\Sigma', \rhd', A')$ *are two transition systems and let* α *be a map from* Σ' *to* Σ. *Assume that* $I \subseteq \Sigma$ *is a set of states such that* $R \subseteq I$; *and let* $I' \subseteq \Sigma'$ *be any iet of states. If*

- $\alpha(A') \subseteq A$,
- $A' \subseteq I'$,
- ${}_{\alpha^{-1}(I)}|\ \rhd'(I') \subseteq I'$,
- $\alpha({}_{\alpha^{-1}(I)\cap I'}|\ \rhd') \subseteq \trianglerighteq$,

then $\alpha([A']\rhd') \subseteq [A]\trianglerighteq$. *Hence any safety property* Γ *invariant to repetition of states can be inherited – i. e, if* P*-executions have property* $\Gamma([A]\rhd \subseteq \Gamma)$, *then all (fair, unfair, or finite)* P'*-executions have safely property* $\alpha^{-1}(\Gamma)$.

Example B (revisited). The most natural choice of α we given at page 13:

$$\begin{array}{lcl} loc_i \circ \alpha & \equiv & (\text{at } read_i) \rightarrow inc_i \ \Box\ loc_i' \\ a \circ \alpha & \equiv & a' \\ x_i \circ \alpha & \equiv & x_i' \end{array}$$

The only property of P needed in the proof is mutual exclusion

$$I: \quad \neg((\text{at } inc_1) \wedge (\text{at } inc_2))$$

$\alpha^{-1}(I)$ expresses almost the same, namely

$$\alpha^{-1}(I): \quad \neg[((\text{at } read_1') \vee (\text{at } inc_1')) \vee ((\text{at } read_2') \vee (\text{at } inc_2'))]$$

The relationship $x_i' = t'$ may now be expressed by a relative invariant

$$I': \quad (\text{at } inc_i') \Rightarrow (x_i' = t')$$

We show that all conditions in the inheritance technique are satisfied.

We assume that $R \subseteq I$ has been proved. It is easily checked that also $\alpha(A') \subseteq A$ and $A' \subseteq I'$ hold.

The relative invariance of I' is proven thus: when (at inc'_i) gets established, $(x'_i = t')$ is also established; and the only action different from $x'_i := t' + 1$ which can falsify $(x'_i = t')$ is the action $t' := x'_{\hat{i}}$ where $\hat{1} = 2$ and $\hat{2} = 1$; but using $\alpha^{-1}(I)$ we get that control cannot simultaneously reside at inc'_i and at $read'_{\hat{i}}$.

We once again use a schema to demonstrate that α maps ${}_{\alpha^{-1}(I)\cap I'} \mid \rhd'$ into $\trianglerighteq$. Now we can draw restrictions both from $\alpha^{-1}(I)$ and from I'.

action	cases	I'	$\alpha^{-1}(I)$	maps to	final state	maps to	image
$wait'_i$	$a' = n$ $[n > 0]$			$a = n$ $wait_i$	$a' = n - 1$ $read'_i$	$a = n - 1$ inc_i	$wait_i$
$read'_i$				inc_i	$t' = n$ inc'_i	inc_i	EMPTY
inc'_i	$x'_i = n$ $t' = m$	$[n = m]$ [1]		$x_i = n$ inc_i	$x'_i = n + 1$ $signal'_i$	$x_i = n + 1$ $signal_i$	inc_i
$signal'_i$	$a' = n$			$a = n$ $signal_i$	$a' = n + 1$ $wait'_i$	$a = n + 1$ $wait_i$	$signal_i$

The rows for $wait'$ and $signal'$ are the same as before. The row for inc' just has the restriction $[n = m]$ in another row than the previous schema. The row for $read'$ is a bit simpler than before it is freed for the argument that the resulting state does not map to $\perp$.

1. Now $[n = m]$ is concluded from the relative invariant, not from the inherited property.

Note that $\alpha^{-1}(I)$ is not used in the schema. It only serves to prove the relative invariance of I'.

5 Two applications

The technique is demonstrated on two slightly more involved examples. The schematic notation introduced in the earlier examples will again be used to prove that α maps ${}_{\alpha^{-1}(I)\cap I'} \mid \rhd'$ into $\trianglerighteq$.

5.1 Peterson's algorithm

Peterson's algorithm [12] provides fair mutual exclusion between two processes p_1 and p_2. We will show that the algorithm is just an encoding of a simple idea: assume p_1 wants to enter its critical section; if p_2 is within its critical section or is waiting with priority, then p_1 must wait until p_2, upon exit, explicitly gives p_1 priority; otherwise p_1 immediately gets priority and can enter. This idea is shown in program P below. Actions inside and outside the critical section are not shown. This is deliberate; they can be inserted in a subsequent refinement of the program.

$pri \in \{1, 2\}$;
cobegin
  repeat
    $uncr_1$: < **if** (at $uncr_2$) **then** $pri := 1$ >;
    $wait_1$: < **await** $pri := 1$ >;
    $crit_1$: < $pri := 2$ >;
  forever
‖**repeat**
    $uncr_2$: < **if** (at $uncr_1$) **then** $pri := 2$ >;
    $wait_2$: < **await** $pri := 2$ >;
    $crit_2$: < $pri := 1$ >;
  forever
coend

Note that each process reads the current value of the other process' program counter. The meaning of the **await**-primitive is that control can only proceed to $crit_i$ if the test is satisfied.

In P it is trivial to prove the following property invariant:

$$I: \quad (\text{at } crit_i') \Rightarrow (pri = i)$$

From this mutual exclusion immediately follows since pri cannot simultaneously take on both values 1 and 2.

The implementation P' of P is obtained by finding another representation of the state space.

boolean $try'_1, try'_2 (try'_1 = try'_2 = false); turn' \in \{1,2\};$
cobegin
repeat
$uncr'_1$: $<$ $try'_1 := true;$
$turn' := 2 >;$
$wait'_1$: $<$ **await** $\neg(try'_2 \wedge turn' = 2 >;$
$crit'_1$: $< try'_1 := false >;$
forever
$\|$**repeat**
$uncr'_2$: $<$ $try'_2 := true;$
$turn' := 1 >;$
$wait'_2$: $<$ **await** $\neg(try'_1 \wedge turn' = 1 >;$
$crit'_2$: $< try'_2 := false >;$
forever
coend

A correspondence between P' and P can be established be the following map α (expression $i : try'_i$: means the value of i such that try_i holds):

$$\begin{array}{lcl} loc_i \circ \alpha & \equiv & loc'_i \\ pri \circ \alpha & \equiv & (try'_1 = try'_2) \rightarrow turn' \,\square\, i : try'_i \end{array}$$

The intuition behind the definition is (again $\hat{1} = 2$ and $\hat{2} = 1$): Process i has priority if it is waiting and process $\hat{\imath}$ is not (so try'_i and not $try'_{\hat{\imath}}$ hold) or if both processes wait and $turn' = i$.

We use the inheritance technique to inherit properties from P (e.g. mutual exclusion). A characterisation of R' is needed for this. The characterisation $I' = I'_1 \wedge I'_2$ consists of the following two properties

$$I'_1 : \quad (\text{at } uncr'_i) \Leftrightarrow \neg try'_i \quad \text{and} \quad I'_2 : \quad try'_i \wedge \neg try'_{\hat{\imath}} \Rightarrow turn' = \hat{\imath}$$

The first property just relates the value of try'_i to the program counter loc'_i and is easily proven invariant. The second property expresses a more subtle relationship used to find the value of pri in $\alpha(t')$ when (s', t') is a transition from $crit'_i$ to $uncr'_i$. Obviously $A' \subseteq I'_2$. The relative invariance of I'_2 is proved next.

The antecedent can be established in two ways. First the action at $uncr'_i$ can establish try'_i; but this simultaneously establishes $turn' = \hat{\imath}$. Second the

action at $crit'_i$ can establish $\neg try'_i$; but assume (at $crit'_i$) $\wedge try'_i$ holds in s'; from I'_1 variable try'_i is true and hence $try'_1 = try'_2$ holds in s'; so pri has the same value in $\alpha(s')$ as $turn'$ has in s'; consequently I implies $turn' = \hat{\imath}$ in s' and hence also in t'. Finally the consequent in I'_2 can be falsified only by the action at $uncr'_i$ but this action simultaneously falsifies $\neg try'_i$.

In order to use the inheritance technique it only remains to be shown that $\alpha(A') \subseteq A$ and that α maps ${}_{\alpha^{-1}(I)\cap I'} \mid \rhd'$ into $\trianglerighteq$. That α maps A' into A is seen by inspection of α. The proper mapping of ${}_{\alpha^{-1}(I)\cap I'} \mid \rhd'$ is demonstrated by the following schema. The program is symmetric in the two processes so only the argument for process 1 is given. Note that, for the fist time, the analysis must be split into cases. This is because we must use two different branches of the α-definition for pri according to the value of try'_2.

action	cases	$I'_1 \wedge I'_2$	$\alpha^{-1}(I)$	maps to	final state	maps to	image
$uncr'_1$	$\neg try'_2$	$uncr'_2$		$uncr_1$ $uncr_2$ (1)	$wait'_1$ $turn' = 2$ try'_1	$wait'_1$ $pri = 1$	$uncr'_1$
	try'_2	$\neg uncr'_2$		$uncr_1$ $\neg uncr_2$ $pri = 2$	$wait'_1$ $turn' = 2$ try'_1	$wait'_1$ $pri = 2$	$\neg uncr'_1$
$wait'_1$	$\neg try'_2$	try'_1		$wait_1$ $pri = 1$	$crit'_1$	$crit'_1$	$wait_1$
	(2) try'_2 $turn' = 1$	try'_1		$wait_1$ $pri = 2$	$crit'_1$	$crit_1$	$wait_1$
$crit'_1$	$\neg try'_2$	try'_1 $turn' = 2$ (3)		$crit_1$ $pri = 1$	$uncr'_1$ $\neg try'_1$	$uncr_1$ $pri = 2$ (4)	$crit_1$
	try'_2			$crit_1$	$uncr'_1$ $\neg try'_1$	$uncr_1$ $pri = 2$	$crit_1$

Four entries deserve a comment.

1. I'_1 gives $\neg try'_1$; so $pri \circ \alpha$ has the value i for which try'_i is true.
2. Since the action at $wait'_1$ can only execute when $\neg try'_2$ or $turn' = 1$ hold, it is sufficient to treat the cases $\neg try'_2$ and $try'_2 \wedge (turn' = 1)$.
3. I'_1 gives the truth of try'_1; that $(turn' = 2)$ holds then follows from I'_2.
4. Since $try'_1 = try'_2$ the value of pri equals the value of $turn'$.

This concludes the argument and we may inherit properties from P. Mutual exclusion in P immediately implies mutual exclusion in P' – i.e. $I \circ \alpha$ reduces to $\neg((\text{at } crit'_1) \wedge (\text{at } crit'_2))$. To show fairness in P' it is necessary to use the

safety property that if $\neg(try'_2 \wedge turn' = 2)$ holds when control reaches $wait'_1$, then $\neg(try'_2 \wedge turn' = 2)$ will continue to hold as long as control remains at $wait'_1$. This is easily deduced from the corresponding P-property that $pri = 1$ continues to hold as long as control remains at $wait_1$.

The atomic actions in P' are coarser-grained than necessary. All actions can be split into atomic reads and writes of the shared variables. The **await**-actions are implemented by **while**-loops using busy waiting. E.g. the $wait'_1$-action refines into

$$\textbf{while} < try''_2 > \wedge < turn'' = 2 > \textbf{do};$$

This construction is not terribly easy to reason about since it hides a temporary variable used to store the value read from try''_2. For transparency we substitute this loop by a **goto**-construction which explicitly uses the temporary variable.

boolean $try''_1, try''_2(try''_1 = try''_2 = false); turn'' \in \{1, 2\}$:
cobegin
  repeat
    $uncr''_1$: $< try''_1 := true >$;
    ent''_1: $< turn'' := 2 >$;
    $wait''_1$: $< temp''_1 = try''_2 >$;
    $test''_1$: $<$ **if** $temp''_1 \wedge turn'' = 2$ **goto** $wait''_1 >$;
    $crit'_1$: $< try''_1 := false >$;
  forever
‖**repeat**
    $uncr''_2$: $< try''_2 := true$;
    ent''_2: $< turn'' := 1 >$;
    $wait''_2$: $< temp''_2 := try''_1 >$;
    $test''_2$: $<$ **if** $temp''_2 \wedge turn'' = 1$ **goto** $wait''_2 >$;
    $crit''_2$: $< try''_2 := false >$;
  forever
coend

We construct α such that the state change caused by the $uncr''$-actions are suppressed and such that process i proceeds to $crit'_i$ in P' when it has sufficient information to pass the **goto**-construction. Hence, if control resides at $test''_i$ and $temp''_i$ is false, then control resides at $crit'_i$ in the image state.

$$
\begin{array}{lcl}
loc'_i \circ \alpha & \equiv & (\text{at } ent''_i) \rightarrow uncr' \\
 & \Box & (\text{at } test''_i) \wedge temp''_i \rightarrow wait' \\
 & \Box & (\text{at } test''_i) \wedge \neg temp''_i \rightarrow crit' \\
 & \Box & loc''_i \\
try'_i \circ \alpha & \equiv & (\text{at } ent''_i) \rightarrow false \ \Box \ try''_i \\
turn' \circ \alpha & \equiv & turn''
\end{array}
$$

This α clearly maps A'' to A'. To prove that the $uncr''_i$-action maps to the empty action and the ent''_i-action to the ent'_i-action we need the following property

$$I'' : \quad ((\text{at } uncr''_i) \Rightarrow \neg try''_i) \wedge ((\text{at } ent''_i) \Rightarrow try''_i)$$

It is easily proven invariant in P''.

The schema below proves that α maps ${}_{I''} \mid \rhd''$ into $\trianglerighteq'$. Note that inherited properties are not used at all in this proof – neither to prove I'' relative invariant nor to prove correct mapping of ${}_{R''} \mid \rhd''$.

action	cases	I''	$\alpha^{-1}(I'')$	maps to	final state	maps to	image
$uncr''_i$		$\neg try''_i$		$uncr'_i$ $\neg try'_i$	ent''_i try''_i	$uncr'_i$ $\neg try'_i$	EMPTY
ent''_i		try''_i		$uncr'_i$ $\neg try'_i$	$wait''_i$ $turn'' = \hat{\imath}$	$wait'_i$ $\neg try'_i$ $turn'' = \hat{\imath}$	$uncr'_i$
$wait''_1$	try''_i			$wait'_i$	$test''_i$ $temp''_i$	$wait'_i$	EMPTY
	$\neg try''_i$			$wait'_i$ $\neg try'_i$	$test''_i$ $\neg temp''_i$	$crit'_i$	$wait'_i$
$test''_i$	$temp''_i$ $turn'' = \hat{\imath}$			$wait'_i$	$wait''_i$	$wait'_i$	EMPTY
	$temp''_i$ $turn'' = i$ $\neg temp''_i$			$wait'_i$ $crit'_i$	$crit''_i$ $crit''_i$	$crit'_i$ $crit'_i$	$wait'_i$ EMPTY
$crit''_i$				$uncr'_i$	$uncr''_i$ $\neg try''_i$	$uncr'_i$ $\neg try'_i$	$uncr'_i$

Now safety properties can me lnnerited. Mutual exclusion in P' was expressed by

$$I' : \ \neg((\text{at } crit'_1) \wedge (\text{at } crit'_2))$$

The property $I' \circ \alpha$ reduces to

$$\begin{array}{c}
\neg(((\text{at } crit''_1) \vee ((\text{at } test''_1) \wedge \neg temp''_1)) \wedge \\
((\text{at } crit''_2) \vee ((\text{at } test''_2) \wedge \neg temp''_2)))
\end{array}$$

This property may in turn be weakened to the desired

$$\neg((\text{at } crit''_2) \wedge (\text{at } crit''_2))$$

Other properties can be similarly inherited.

Implementing a bounded-size queue by a circular buffer

The following program P is an abstract description of a queue with a maximum capacity of N elements.

sequence q $(q = ())$; **element** e, f;
cobegin
  repeat
    $prod_p$: $< produce(e) >$;
    $wait_p$: $<$ **await** $|q] < N >$;
    $insp$: $< q := q * e >$;
  forever
 ‖**repeat**
    $wait_c$: $<$ **await**$|q| > 0 >$;
    rem_c: $< f, q := head(q), tail(q) >$;
    con_c: $< consume(f) >$;
  forever
coend

Let $\bar{e}$ denote the sequence of messages produced so far and $\bar{f}$ the sequence of messages consumed so far. Then the most important feature of the program is the following safety property Γ which is obviously true for program P.

$$\Gamma : \bar{f} \sqsubseteq \bar{e}$$

The two sequences $\bar{e}$ and $\bar{f}$ may each be specified by giving recursive definitions for finite executions ϕ and by taking the value on an infinite execution to be the limit of $\bar{e}$ respectively $\bar{f}$ on the finite prefixes of ϕ. The recursive definitions are similar for $\bar{e}$ and $\bar{f}$; for $\bar{e}$ it is

$$\begin{array}{lcl} \bar{e}(s) & = & (), \qquad \text{for } s \in \Sigma \\ \bar{e}(\phi * s) & = & \textbf{if } (\text{at } prod_p)(\phi^\bullet) \wedge (\text{at } wait_p)(s) \\ & & \quad \textbf{then } \bar{e}(\phi) * e(s) \textbf{ else } \bar{e}(\phi) \end{array}$$

A property which will be of use in applying the inheritance technique is the following:

$$I' : \ ((\text{at } ins_p) \Rightarrow (|q| < N)) \wedge ((\text{at } rem_c) \Rightarrow (|q| > 0))$$

This property is obviously true for program P.

The elements of the queue may be stored in an array of size $N+1$ by using a circus allocation policy. Two integers i' and j' serve as pointers into the array; they are incremented using modulo $(N+1)$-arithmetic (we use $\oplus$ for addition, $\ominus$ for subtraction in modulo $(N+1)$-arithmetic). Different conventions can be used for the implementation; here we use

$$|q| = k \Leftrightarrow i' \oplus (k+1) = j'$$

Hence q is empty ($q = ()$) when $i' \oplus 1 = j'$ and q is full ($|q] = N$) when $i' = j'$; i' points to the location most recently freed in the array and j' points to the location which will be occupied next.

array $[0..N]$ **of element** $q'(q' = [?, \ldots, ?])$;
integer $i', j'(i' \oplus 1 = j')$; **element** e', f';
cobegin
  repeat
    $prod'_p$: $< produce(e) >$;
    $wait'_p$: $<$ **await** $i' \neq j' >$;
    put'_p: $< q'[j'] := e >$;
    $ins'p$: $< j' := j' \oplus 1 >$;
  forever
$\|$**repeat**
    $wait'_c$: $<$ **await** $i' \oplus 1 \neq j' >$;
    rem'_c: $< i' := i' \oplus 1 >$;
    get'_c: $< f' := q'[i'] >$;
    con'_c: $< consume(f') >$;
  forever
coend

We will map states of P' into states of P and remark that the state space of P' is spanned by $q', i', j', e', f', loc'_p$ and loc'_c, whereas the state space of P is spanned by q, e, f, loc_p and loc_c.

$$
\begin{array}{lcl}
q \circ \alpha & \equiv & (i' \oplus 1 = j') \rightarrow () \ \Box \ (q'[i' \oplus 1], q'[i' \oplus 2], \ldots, q'[j' \ominus 1]) \\
e \circ \alpha & \equiv & e' \\
f \circ \alpha & \equiv & (\text{at } get'_c) \rightarrow q'[i'] \ \Box \ f' \\
loc_p \circ \alpha & \equiv & (\text{at } put'_p) \rightarrow ins_p \ \Box \ loc'_p \\
loc_c \circ \alpha & \equiv & (\text{at } get'_c) \rightarrow con_c \ \Box \ loc'_c
\end{array}
$$

Note that the definition of $q \circ \alpha$ gives the property $|q| = k \Leftrightarrow (i' \oplus (k+1)) = j'$.

To use the inheritance technique it is necessary to introduce a P'-specific property assuring that the ins'_p-action will map to the ins_p-action.

$$I' : \ (\text{at } ins'_p) \Rightarrow q'[j'] = e'$$

It is a trivial fact that this implication holds; $q'[j'] = e'$ can only be falsified by changing j', g', or e'; and only the producer can do this.

It is easily checked that $\alpha(A') \subseteq A$.

An important point in the proof that α maps ${}_{\alpha^{-1}(I) \cap I'} \mid \rhd'$ into $\trianglerighteq$ is to show that the action $q'[j'] := e'$ maps into the empty action. The image action could in principle change f since f is defined to have value $q'[i']$ when $(\text{at } get'_c)$ holds. The inherited invariant $\alpha^{-1}(I)$ is used to deduce that this does not happen. The entire proof is given in the following schema. I, J, E, F, and Q are parameter values.

action	cases	I'	$\alpha^{-1}(I)$	maps to	final state	maps to	image
$prod'_i$				$prod_p$	$wait'_p$ $e' = E$	$wait_p$ [1] $e = E$	$prod_p$
$wait'_p$	$i' \neq j'$			$wait_p$ [2] $\lvert q \rvert < N$	put'_p	ins_p	$wait_p$
put'_p	$j' = J$		$i' \neq j$ [3]	ins_p	ins'_p $q'(J) = e'$	ins_p [4]	EMPTY
ins'_p	$j' = J$	$q'[J] = e'$	$i' \neq j$	ins'_p $q = Q$	$prod'_p$ $j' = J \oplus 1$	$prod_p$ [5] $q = Q * e$	ins_p
$wait'_c$	$i' \oplus 1 \neq j'$			$wait_c$ $\lvert q \rvert > 0$	rem'_c	rem_c	$wait_c$
rem'_c	$i' = I$ $F = q'[I \oplus 1]$		$I \oplus 1 \neq j$	rem_p [6] $q = F * Q$	get'_c $i' = i \oplus 1'$	con_c [7] $q = Q$ $f = F$	rem_c
get'_c	$F = q'[i' \oplus 1]$			con_c $f = F$	con'_c $f' = q'[i']$	con_c $f = F$	EMPTY
con'_c				con_c	$wait'_c$	$wait_c$	con_c

1. It is assumed that $produce(e')$ has same semantics as $produce(e)$; hence the same value E is produced.

2. Since $|q| = k \Leftrightarrow i' \oplus (k+1) = j'$, it follows that $i' \neq j'$ implies $|q| < N$.

3. (at ins_p) $\Rightarrow$ ($|q| < N$) holds in P. Using $|q| = k \Leftrightarrow i' \oplus (k+1) = j'$ this gives $i' \neq j'$.

4. Only entry J of q' is changed; since $J \neq i'$ and i' remains unchanged we get that f does not change.

5. In t' we have $i' \oplus 1 \neq J \oplus 1 = j'$; hence q has form $(q'[i' \oplus 1], q'[i' \oplus 2], \ldots, q'[j' \ominus 1])$ in $\alpha(t')$. From $j' = J \oplus 1$ and $e = e' = q'[J]$ we get that $q = Q * e$.

6. From $i' \oplus 1 \neq j'$ we get $|q| > 0$; thus we can write q as $F * Q = (q'[I \oplus 1], \ldots, q'[j' \ominus 1])$.

7. No element in q' has changed; hence q has form $(q'[I \oplus 2], \ldots, q'[j' \ominus 1])$, or () if $i' \oplus 1 = j'$. Furthermore f is defined as $q'[i'] = q'[I \oplus 1] = F$ when the consumer is at get'_c.

Now the property $\bar{f} \sqsubseteq \bar{e}$ can be inherited. We will show that $(\bar{f} \sqsubseteq \bar{e}) \circ \alpha$ is the same function as $\bar{f}' \sqsubseteq \bar{e}'$ where $\bar{e}'$ and $\bar{f}'$ are given by recursive definitions similar to those for $\bar{e}$ and $\bar{f}$. It suffices to show that each of $\bar{f} \circ \alpha$ and $\bar{e} \circ \alpha$ satisfy the same recursive definition as $\bar{f}'$ and $\bar{e}'$ respectively. For $\bar{e} \circ \alpha$ this is shown thus

$$
\begin{array}{lcl}
\bar{e} \circ \alpha(s') & \equiv & \bar{e}(\alpha(s')) = (), \quad \text{for } s' \in \Sigma. \\
\bar{e} \circ \alpha(\phi' * s') & \equiv & \bar{e}(\alpha(\phi') * \alpha(s')) \\
& \equiv & \textbf{if } (\text{at } prod_p)(\alpha(\phi')) \wedge (\text{at } wait_p)(\alpha(s')) \\
& & \quad \textbf{then } \bar{e}(\alpha(\phi')) * e(\alpha(s')) \textbf{ else } \bar{e}(\alpha(\phi')) \\
& \equiv & \textbf{if } (\text{at } prod'_p)(\phi'^{\bullet}) \wedge (\text{at } wait_p)(s') \\
& & \quad \textbf{then } \bar{e} \circ \alpha(\phi') * e'(s') \textbf{ else } \bar{e} \circ \alpha(\phi')
\end{array}
$$

A similar calculation shows $\bar{f} \circ \alpha = \bar{f}'$. Hence the property $\bar{f}' \sqsubseteq \bar{e}'$ holds for P'.

6 History defined maps

In the second refinement of Peterson's algorithm we have constructed β so that it suppresses state changes in Σ' until appropriate state changes take

place in Σ''. This was possible because overwritten values of the variables try_1 and try_2 could be deduced from the current state in P''. Hence the current state s'' was sufficient to define the state $\beta(s'')$.

Sometimes it is not possible to deduce overwritten values from the current state. Then we need a more general approach where α also depends on previous states. We say that α is *history defined* if α is a function from Σ'^+ to Σ. To distinguish history defined maps from ordinary state-to-state maps we call the latter *state defined* maps.

A history defined map a induce a map $\tilde{\alpha}$ from $\Sigma'^+ \cup \Sigma'^\omega$ to $\Sigma^+ \cup \Sigma^\omega$ by the definition:

1. $\tilde{\alpha}(s') = \alpha(s')$ for $s' \in \Sigma'$,
2. $\tilde{\alpha}(\phi' * s') = \tilde{\alpha}(\phi') * \alpha(\phi' * s')$ for $s' \in \Sigma'$ and $\phi' \in \Sigma'^+$,
3. $\tilde{\alpha}(\phi') = \text{lub}\{\tilde{\alpha}(\psi') \mid \psi' \sqsupset \phi'\}$ for $\phi' \in \Sigma'^\omega$ (defined because $\tilde{\alpha}$ is monotonic on finite ψ')

The map $\tilde{\alpha}$ can be used to inherit properties. This more complicated however, than for state defined maps. We once again need

Proposition. *If $\Gamma$is a safety property, then also $\tilde{\alpha}^{-1}(\Gamma)$ is.*

Proof. We must show that $\tilde{\alpha}^{-1}(\Gamma)$ is prefix closed and ω-complete. Let $\phi' \in \tilde{\alpha}^{-1}(\Gamma)$ and let $\psi' \sqsubseteq \phi'$. By the definition of $\tilde{\alpha}$, $\psi' \sqsubseteq \phi'$ implies $\tilde{\alpha}(\psi') \sqsubseteq \tilde{\alpha}(\phi')$. As Γ is prefix closed, we get $\tilde{\alpha}(\psi') \in \Gamma$; hence also $\psi' \in \tilde{\alpha}^{-1}(\Gamma)$.
Now assume $\phi'_1 \sqsubseteq \phi'_2 \sqsubseteq \ldots$ is a chain in $\tilde{\alpha}^{-1}(\Gamma)$ with least upper bound ϕ'. Then $\tilde{\alpha}(\phi'_1) \sqsubseteq \tilde{\alpha}(\phi'_2) \sqsubseteq \ldots$ is a chain which, by definition, has least upper bound $\tilde{\alpha}(\phi')$. Since $\tilde{\alpha}(\phi'_i)$ belongs to Γ for all i and Γ is ω-complete, also $\tilde{\alpha}(\phi')$ belongs to Γ; hence $\phi' \in \tilde{\alpha}^{-1}(\Gamma')$.

In general it may be difficult to find a simple expression of an inherited property. If Γ is defined by F_Γ, then $\tilde{\alpha}^{-1}(\Gamma)$ is defined by $F_\Gamma \circ \tilde{\alpha}$ where both F_Γ and $\tilde{\alpha}$ may refer to prefixes of their arguments.

In a special case $F_\Gamma \circ \tilde{\alpha}$ may be reduced as in the previous sections. This happens if F_Γ only refers to variables $v_1, \ldots, v_n$ and $v_i \circ \alpha$ for these variables is defined by

$$v_i \circ \alpha(\phi') = v_i \circ \beta(\phi'^\bullet)$$

for some $\beta : \Sigma' \to \Sigma$. Then $F_\Gamma \circ \tilde{\alpha}$ is nothing more than the usual function $F_\Gamma \circ \beta$.

The inheritance technique carries over to history defined maps rather smoothly. For completeness the technique is generalised such that the reachable states in $[A']\rhd'$ may be characterised as terminal states of sequences in the intersection of two *general* safety properties, $\tilde{\alpha}^{-1}(\Gamma)$ and Γ'. In some cases the set of states in property $\tilde{\alpha}^{-1}(\Gamma) \cap \Gamma'$ will be strictly less than the intersection of the set of states in property $\tilde{\alpha}^{-1}(\Gamma)$ with the set of states in property Γ'.

General inheritance technique. *Let $P = (\Sigma, \rhd, A)$ and $P' = (\Sigma', \rhd', A')$ be two transition systems and let α be a map from Σ'^+ to Σ. Assume sequences in $[A]\rhd$ have safety property Γ and let Γ' be any safety property over Σ'. If*

- $\alpha(A') \subseteq A$,
- $A' \subseteq \Gamma'$
- *for all $\phi' \in \tilde{\alpha}^{-1}(\Gamma) \cap \Gamma'$ and $(\phi'^\bullet, s') \in \rhd'$ the following holds*
  - **a.** $\phi' * s' \in \Gamma'$,
  - **b.** $\alpha(\phi', \phi' * s') \in \trianglerighteq$

then $\tilde{\alpha}([A']\rhd') \subseteq [A]\rhd$. Hence any safety property Γ_0 invariant to repetition of states can be inherited - i.e. if $[A]\rhd \subseteq \Gamma_0$, then $[A']\rhd' \subseteq \tilde{\alpha}^{-1}(\Gamma_0)$.

Proof. We prove by induction on the length of ϕ' that $\phi' \in [A']\rhd'$ implies $\phi' \in \Gamma'$ and $\tilde{\alpha}(\phi') \in [A]\rhd$.
Basis: $\tilde{\alpha}(A') = \alpha(A') \subseteq A \subseteq [A]\rhd$
Induction step: Let $\phi' \in [A']\rhd'$. The induction hypothesis gives $\phi' \in \Gamma'$ and $\phi' \in \tilde{\alpha}^{-1}([A]\rhd) \subseteq \tilde{\alpha}^{-1}(\Gamma)$. Let $(\phi'^\bullet, s')$ be a transition in $\rhd'$; then **a.** gives $\phi' * s' \in \Gamma'$; since $\tilde{(}\phi' * s') = \tilde{\alpha}(\phi') * \alpha(s')$ and $\tilde{\alpha}(\phi')^\bullet = \alpha(\phi')$, we get from **b.** and $\tilde{\alpha}(\phi') \in [A]\trianglerighteq$ that $\tilde{\alpha}(\phi' * s') \in [A]\trianglerighteq$.

Note, if $\alpha : \Sigma'^{+} \to \Sigma$ is defined from just the current state – $\alpha(\phi') = \beta(\phi'^{\bullet})$ for some $\beta : \Sigma' \to \Sigma$ – and if Γ and Γ' are just defined by sets of states I and I', then the general technique reduces to the previously introduced technique.

Example. We now present another implementation of a bounded-size queue by a circular buffer. In the new implementation we use the convention

$$|q| = k \Leftrightarrow i' \oplus k = j'$$

(Again $\oplus$ denotes addition modulo $N + 1$.)
The pointers i' and j' are used such that i' points to the element to be removed next and j' points to the location where the next insertion will take place.

array $[0..N]$ **of element** $q'(q' = [?, \ldots, ?])$;
integer $i', j'(i' = j')$; **element** $e'f'$;
cobegin
  repeat
    $prod'_p$: $< produce(e') >$;
    $wait'_p$: $<$ **await** $i' \oplus N \neq j' >$;
    put'_p: $< q'[j'] := e' >$;
    $ins'p$: $< j' := j' \oplus 1 >$;
  forever
‖**repeat**
    $wait'_c$: $<$ **await** $i' \neq j' >$;
    get'_c: $< f' := q'[i'] >$;
    rem'_c: $< i' := i' \oplus 1 >$;
    con'_c: $< consume(f') >$;
  forever
coend

The history defined map $\alpha : \Sigma^{+} \to \Sigma$ is once again defined by exhibiting the function $v \circ \alpha$ where v is a variable of P. The prime motivation for introducing a history defined map is to suppress state changes. After some variable v' changes its value we want to use the value before the state change. This is accomplished by the function $prev[b, v']$ where p is a predicate on states in Σ'. It takes a sequence $\phi' \in \Sigma'^{+}$ and returns the value of v' in the last

state of ϕ' where p was true. Note that p must hold at some previous point in ϕ' in order for $prev[p, v']$ to be defined. Unless the $prev[p, v']$-function is used, all functions are to be evaluated in the final state $\phi'^{\bullet}$ of an argument ϕ'.

$$\begin{array}{lcl} q \circ \alpha & \equiv & i' = j' \rightarrow () \ \Box \ (q'[i'], \ldots, q'[j' \ominus 1]) \\ e \circ \alpha & \equiv & e' \\ f \circ \alpha & \equiv & (\text{at } rem'_c) \rightarrow prev[(\text{at } get'_c), f'] \ \Box \ f' \\ loc_p \circ \alpha & \equiv & (\text{at } put'_p) \rightarrow ins_p \ \Box \ loc'_p \\ loc_c \circ \alpha & \equiv & (\text{at } get'_c) \rightarrow con_c \ \Box \ loc'_c \end{array}$$

The only place where history plays a role is in the definition of $f \circ \alpha$. The definition says that if control is at rem'_c, then the previous value of f' should be used – not the current.

We will use the general inheritance technique to show that $\tilde{\alpha}$ maps $[A']\rhd'$ into $[A]\trianglerighteq$. The two safety properties Γ and Γ' are the sets of sequences with states in I and I' respectively given by

$$I \ : \ ((\text{at } ins_p) \Rightarrow |q| < N) \wedge ((\text{at } rem_c) \Rightarrow |q| > 0)$$

$$I' \ : \ ((\text{at } ins'_p) \Rightarrow q'|j'| = e') \wedge ((\text{at } rem'_c) \Rightarrow f' = q'[i'])$$

The first two items of the technique, $\alpha(A') \subseteq A$ and $A' \subseteq I'$, are easily checked. Item **a.** requires just a proof of relitive invoice of I'. That (at ins'_p) implies $q'[j'| = e'$ follows from the fact that only the producer can change $j', e', q'[j']$ or the producer's program location. To prove that (at rem'_c) implies $f' = q'[i']$ we need to establish that the producer cannot change $q'[i']$ when the consumer is at rem'_c. This follows from the inherited property

$$(\text{at } rem'_c) \Rightarrow i' \neq j'$$

For the last item in the technique we once again use a schema. Now the image of $(\phi', \phi' * s')$ for transition $\phi'^{\bullet}, s')$ in $\rhd'$ may not only be determined by $\phi'^{\bullet}$ and s' but also by ϕ-states before $\phi'^{\bullet}$. The column "cases" should be used to distinguish between such cases.

action	cases	I'	$\alpha^{-1}(I)$	maps to	final state	maps to	image
$prod'_p$				$prod_p$	$\begin{matrix} wait'_p \\ e' = E \end{matrix}$	$\begin{matrix} wait_p \\ e = E \end{matrix}$	$prod_p$
$wait'_p$	$i' \oplus N \neq j'$			$\begin{matrix} wait_p \\ \lvert q \rvert < N \end{matrix}$	put'_p	ins_p	$wait_p$
put'_p	$j' = J$			ins_p	$\begin{matrix} ins'_p \\ q'[J] = e' \end{matrix}$	ins_p [1]	EMPTY
ins'_p	$\begin{matrix} j' = J \\ e' = E \end{matrix}$	$q'[J] = E$	$i' \oplus N \neq j$	$\begin{matrix} ins_p \\ q = Q \end{matrix}$	$\begin{matrix} prod'_p \\ j' = J \oplus 1 \end{matrix}$	$\begin{matrix} prod_p \\ q = Q * E \end{matrix}$ [2]	ins_p
$wait'_c$	$i' \neq j'$			$\begin{matrix} wait_c \\ \lvert q \rvert > 0 \end{matrix}$	get'_c	rem_c	$wait_c$
get'_c	$f' = F$			$\begin{matrix} rem_c \\ f = F \end{matrix}$	$\begin{matrix} rem'_c \\ f' = q'[i'] \end{matrix}$	$\begin{matrix} rem_c \\ f = F \end{matrix}$ [3]	EMPTY
rem'_c	$\begin{matrix} i' = I \\ f' = F \end{matrix}$	$F = q'[I]$	$I \neq j'$	$\begin{matrix} rem_c \\ q = F * q \end{matrix}$	$\begin{matrix} con'_c \\ i' = I \oplus 1 \end{matrix}$	$\begin{matrix} con_c \\ q = Q \\ f = F \end{matrix}$ [4]	rem_c
con'_c				con_c	$wait'_c$	$wait_c$	con_c

1. $q'[J]$ has changed, but the value of $q'[J]$ is not used in the definition of $q \circ \alpha$.

2. From $\alpha^{-1}(I)$ we get $i' \neq j'$ in the final state. Hence q is not empty, and as j' has been increased by one, the element $q'[J] = E$ has been inserted.

3. f is set to the previous value of f when control is at rem'_c.

4. At con'_c the value of f is not history defined any more. It thus has the value $q'[I]$ as desired.

It should be noted that in opposition to the previous implementation we do not have to argue that the put'_p-action leaves f unchanged in P; since f is history defined, $q'[i']$ does not occur in the α-definition for f.

We can now infer that function $(\bar{f} \sqsubseteq \bar{e}) \circ \tilde{\alpha}$ yields true when applied to sequences in $[A']\triangleright'$. We prove again that $(\bar{f} \sqsubseteq \bar{e}) \circ \tilde{\alpha}$ is the same function as $\bar{f}' \sqsubseteq \bar{e}'$ by proving $\bar{e} \circ \alpha = \bar{e}'$ and $\bar{f} \circ \alpha = \bar{f}'$. The case $\bar{f} \circ \alpha = \bar{f}'$ is the more involved so we exhibit the reasoning for this case. $\bar{f}$ is recursively defined by

$$\begin{array}{lcl} \bar{f}(s) & = & (\,) \\ \bar{f}(\phi * s) & = & \textbf{if } (\text{at } con_c)(\phi'^{\bullet}) \textbf{ and } (\text{at } wait_p)(s) \\ & & \quad \textbf{then } \bar{f}(\phi) * f(s) \textbf{ else } \bar{f}(\phi') \end{array}$$

We now show that $\bar{f} \circ \tilde{\alpha}$ matches the similar recursive definition of $\bar{f}'$.

$\bar{f} \circ \tilde{\alpha}(s') = \bar{f}(\tilde{\alpha}(s')) = ()$, since $\tilde{\alpha}(s')$ is a one-element sequence,

$$\begin{array}{lcl} \bar{f} \circ \tilde{\alpha}(\phi' * s') & = & \bar{f}(\tilde{\alpha}(\phi') * \alpha(\phi' * s')) \\ & = & \textbf{if } (\text{at } con_c)(\tilde{\alpha}(\phi')^\bullet) \textbf{ and } (\text{at } wait_c)(\alpha(\phi' * s')) \\ & & \quad \textbf{then } \bar{f}(\tilde{\alpha}(\phi')) * f(\alpha(\phi' * s')) \textbf{ else } \bar{f}(\tilde{\alpha}(\phi')) \\ & = & \textbf{if } (\text{at } con'_c)(\phi'^\bullet) \textbf{ and } (\text{at } wait_c)(s')) \\ & & \quad \textbf{then } \bar{f} \circ \tilde{\alpha}(\phi') * f'(s') \textbf{ else } \bar{f} \circ \tilde{\alpha}(\phi') \end{array}$$

The reduction of $f(\alpha(\phi' * s'))$ to $f'(s')$ is possible because s' does not satisfy $(\text{at } rem_c)$.
In conclusion we then get that $\bar{f}' \sqsubseteq \bar{e}'$ is true in $[A']\triangleright'$.

7 Completeness

We have presented a proof technique which allows us to inherit safety properties. A natural question then arises: is our proof technique complete? Before this question is answered we have to discuss what completeness means.

In [1], [11], and [6] the authors all deal with open systems. This means that it is possible to compare the external behaviour of a program at its interface to the corresponding external behaviour of the intended implementation. A completeness result in their approaches then says that it is possible to make an implementation proof by their methods if it turns out that the external behaviour of the intended implementation actually implements the external behaviour of the program.

In our approach the situation is radically different. We deal only with closed systems so there is no interface at which the behaviours can be compared. Instead the chosen α shows how to relate high level behaviour to low level behaviour. This means that α is not only part of the proof technique but also part of the interpretation of the relationship between two levels. Consequently we cannot give a criterion which is independent of α for what it means for one system to implement another system. Instead our completeness result will show that, under very mild restrictions, it is possible to get any safety property of a low level system as an inherited property of any high level system. The history map α only has to be carefully chosen.

> **Completeness theorem.** *Assume that $P = (\Sigma, \triangleright, A)$ is a transition system with both reachable and unreachable states, and as-*

sume that $P' = (\Sigma', \rhd', A')$ *is a transition system with a safety property* Γ'_0 *which is invariant to repetition of states. Then there exist a safety property* Γ_0 *of* P*, a safety property* Γ' *of* P' *and a history defined map* α *such that the conditions in the general inheritance technique are satisfied and such that* $\Gamma'_0 = \tilde{\alpha}^{-1}(\Gamma_0)$.

Proof. We let Γ_0 be the property $[A]\trianglerighteq$ and we let Γ be the property $[A']\trianglerighteq'$. To define α, let s^u be an unreachable state of P and let s^r be a state in A (this set is non-empty because P has reachable states) We then define a by

$$\alpha(\phi') = \begin{cases} s^r, & \text{if } \phi' \in \Gamma'_0 \\ s^u, & \text{if } \phi' \notin \Gamma'_0 \end{cases}$$

Since each sequence $s^r s^r \cdots s^r$ is a member of Γ_0 and since no sequence $\phi \in \Gamma_0$ contains unreachable states we obviously have $\Gamma'_0 = \tilde{\alpha}^{-1}(\Gamma_0)$. So we are left with the proof that Γ_0, Γ', and α satisfy the conditions in the general inheritance technique. Since A' is a subset of Γ'_0 we have $\alpha(A') = \{s^r\} \subseteq A$. By definition we furthermore have $A' \subseteq \Gamma'$. Finally $\phi' \in \tilde{\alpha}^{-1}(\Gamma_0) \cap \Gamma'$ implies $\phi' \in [A']\rhd'$. So if $(\phi'^{\bullet}, s') \in \rhd'$ we have $\phi' * s \in [A']\rhd' = \Gamma$ and since $[A']\rhd' \subseteq \Gamma'_0$ we furthermore have $\alpha(\phi', \phi' * s') = (s^r, s^r) \in \trianglerighteq$.

The apparent strength expressed in the theorem – that properties can be inherited between lost any pair of transition systems – is also a major weakness. It shows how much the map a can be used to code up the differences between P and P'. If the distance between P and P' is too large, then probably α is very difficult to work with and possibly the properties of P will not get into real use.

There is also another obstacle to the theorem. It does not address the problem of reusing already proven properties as characterisations. In fact the property Γ' introduced in the proof cannot in any sense be said to be orthogonal with the inherited property $\tilde{\alpha}^{-1}(\Gamma_0)$ since we actually have that $\Gamma' \subseteq \tilde{\alpha}^{-1}(\Gamma_0)$. So the proof technique in this case includes an outright proof of the invariance of Γ' and there is consequently no need for the technique.

It is not easy to answer the question what it really means for two properties to be orthogonal. Intuitively Γ' and Γ'_0 in the completeness theorem

should be orthogonal in order to ensure that only a minimum effort is put into proving Γ' relatively invariant with respect to Γ'_0. One might suspect that a best choice of Γ' then word be to make Γ' as weak as possible. This is actually feasible since, as is easily seen, if both Γ'_1 and Γ'_2 can serve the same purpose as Γ' in the general inheritance technique, then so can $\Gamma'_1 \cup \Gamma'_2$ so a largest such Γ' exists. Whether the largest Γ' is easier to prove relatively invariant is not clear, however; it may depend on the notation chosen for expressing Γ' and it may occur that a stronger property is easier to express and work with than a weaker property.

8 Discussion

Gradually stronger techniques for inheriting safety properties have been presented. The limit of strength has not been reached, however. The following should be the most general formulation of a technique along the lines in this paper ($\phi \sqsubseteq_P \psi$ means ϕ can be extended to ψ by using zero or more transitions from $\rhd$):

> *Let* $(\Sigma, \rhd, A)$ *and* $(\Sigma', \rhd', A')$ *be two transitions systems and let* α *be a map from* ${\Sigma'}^+$ *to* Σ^+. *Assume* $[A]\rhd \subseteq \Gamma$.
> *If (for some* Γ'*)*
>
> - $\alpha(A') \subseteq [A]\rhd$,
> - $A' \subseteq \Gamma'$,
> - *for all* $(\phi'^\bullet, s') \in \rhd'$ *with* $\phi' \in \alpha^{-1}(\Gamma) \cap \Gamma'$*:*
>   - $\phi' * s' \in \Gamma'$,
>   - $\alpha(\phi') \sqsubseteq_P \alpha(\phi' * s')$,
>
> *then* $\alpha([A']\rhd') \subseteq [A]\rhd$.

This can be proven as before by induction on the length of sequences in $[A']\lhd'$.

Viewed as specialisations the two previous formulations of the technique constrain the function $\alpha : {\Sigma'}^+ \to \Sigma^+$ by $\alpha(\phi' * s') = \alpha(\phi') * \alpha(s')$ respectively $\alpha(\phi' * s') = \alpha(\phi') * \beta(\phi' * s')$ for some $\beta : {\Sigma'}^+ \to \Sigma$; these constraints ease

finding expressions for $\alpha^{-1}(\Gamma)$ and they limit the number of checks to be performed when applying the technique.

The above formulation leaves α totally unconstrained; hence no guidelines can be given for expressing the property $\alpha^{-1}(\Gamma)$. The completeness result says that this stronger technique may only be necessary for pragmatic reasons. So far no programs have been found where more general techniques than those presented previously seem to be helpful. Furthermore, as demonstrated by the examples the proof technique is simplest to use for the first versions of the technique. The practical relevance of the technique is more doubtful when history information needs to be introduced. Whether the introduction of history information is necessary crucially depends on how "close" the abstract and the concrete descriptions are. If they are close, then the technique may be of great value; if not, then it may turn out to be simpler to give a direct proof of the safety properties in question.

As noticed more times, proofs of liveness properties are often based on safety properties. Hence techniques for inheriting safety properties also aid in establishing liveness properties. The liveness properties are not inherited, however. Their proofs are repeated except for the safety parts.

All liveness properties are based on some basic fairness requirements; certain sets of transitions must under certain conditions be chosen infinitely often in infinite executions.

Inheritance of liveness properties without repetition of proofs can indeed be done by demonstrating directly that the fairness requirements are met in all images of execution sequences. But in opposition to the case for safety properties it does not seem possible to use inherited liveness properties in such demonstrations.

This difference seems to be due to the fact that safety properties are proved by induction on the length of execution sequences, whereas liveness properties are established by arguments involving well-foundedness. An inductive argument is used to prove the validity of the inheritance technique for safety properties and this argument does not carry over to liveness properties.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. Proceedings of the 2nd Symposium on Logic in Computer Science, pp. 165-175, Edinburgh, U. K. July 1988.

[2] B. Alpern and F. B. Schneider, Defining Liveness, Information Processing Letters Vol. 21, No. 4, October 1985, pp. 181-185.

[3] B. Alpern, A. J. Demers and F. B. Schneider, Safety without Stuttering. Information Processing Letters, Vol. 23, no. 4, November 1986, pp. 177-180, North-Holland.

[4] E. W. Dijkstra, Cooperating Sequential Processes. In Programming Languages, F. Genuys (Ed.), Academic Press 1968, pp. 43-112.

[5] E. W. Dijkstra et. al., On-the-Fly Garbage Collection: An Exercise in Cooperation. ACM Communications, Vol. 27, No. 11, November 1978, pp. 966-975.

[6] B. Jonsson. Modular Verification of Asynchronous Networks. Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing. Vancouver, Canada, August 1987, pp. 152-166.

[7] L. Lamport, An Assertional Correctness Proof of a Distributed Algorithm. Science of Computer Programming, Vol. 2, No. 3, December 1982, pp. 175-206.

[8] L. Lamport, Specifying Concurrent Program Modules. ACM Transactions on Programming Languages and Systems, Vol. 5, no. 2, April 1983, pp. 190-222.

[9] L. Lamport, What Good is Temporal Logic?. Information Processing 83, pp. 657-667, North-Holland Pub. Co. 1983.

[10] L. Lamport. A simple approach to specifying concurrent systems. Communications of the ACM, 32, 1, January 1989, pp. 32-47.

[11] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 137-151.

[12] G. L. Peterson, Myths about the Mutual Exclusion Problem. Information Processing Letters, Vol. 12, No. 5 1981, pp. 115-116.