TRANSITION LOGIC

how to reason about temporal properties

of programs in a compositional way

Rob Gerth

RUU-CS-83-17

November 1983

Februari 1984

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

TRANSITION LOGIC

how to reason about temporal properties of programs in a compositional way

Rob Gerth

Technical Report-RUU-CS-83-17

November 1983

Februari 1984

Dspartment of Computer Science

University of Utrecht

P.O. Box 80.012

3508 TA Utrecht

the Netherlands

TRANSITION LOGIC

how to reason about temporal
properties in a compositional way

ROB GERTH
Department of Computer Science, University of Utrecht
P.O. Box 80.012, 3508TA Utrecht, the Netherlands

ABSTRACT.

This paper addresses the problem of obtaining formal proof systems that support reasoning about temporal properties of parallel programs in a way that is *compositional* or 'syntax directed' - the distinctive feature of e.g. Hoare style proof systems. Now, temporal properties of programs express properties about execution traces of these programs. In the presence of concurrency, compositionality is obtained by concentrating exclusively on the execution traces associated with the atomic actions of programs - the so-called *transitions* of those programs. To reason about such transitions in a compositional way, 'Transition Logic' is proposed, expressing properties of the form "every transition of a program, say α, that starts in a state satisfying assertion p, must end in a state validating q": $[p]\alpha[q]$. Essential is that these assertions can express properties of control locations of programs, too. For this logic, a compositional proof system is obtained which is proved to be sound and relatively complete. An interesting feature of the proof system is the axiomatization of the flow-of-control of programs. The relevance of this logic is supported by the fact that the temporal behaviour of a program is ultimately provable in terms of properties of its transitions, as shown by the work of Z. Manna and A. Pnueli [16].

TO APPEAR AT THE 16th ACM STOC

1. INTRODUCTION.

The motivation for this research originates from the principle that the development of a program and the verification that it meets its specification should proceed simultaneously and should guide each other.

This paradigm has found forceful proponents in, e.g., E.W. Dijkstra [6] and D. Gries [8]. Whether it is a practical one is debatable. However, the principle is of obvious methodological value and it makes sense to study its consequences for verification techniques.

What are those consequences? Firstly, as the verification effort should guide program development, it should reflect the way in which programs are composed from its (sub-) programs. I.e., the method should be syntax directed. Secondly, high-level languages allow (sub-) programs to be viewed as black boxes: Only the behaviour is relevant, not the way this behaviour is achieved. Hence, verification should be based on specifications of programs only. Consequently, the following principle is implied:

The specification of a program should be verifiable in terms of the specifications of its (syntactic) subprograms.

It is not difficult to discern Frege's notion of *semantic compositionality* in this statement ([11]).

The principle neatly demarcates various approaches to program verification:
(1) For input-output or partial correctness properties of sequential programs, one may contrast Floyd's non-compositional 'inductive assertion method' [7],

---

These investigations were supported by the Foundation for Computer Science Research in the Netherlands (SION) with financial aid from the Netherlands Organization for the Advancement of Pure Research (ZWO).

with Hoare's compositional 'Hoare logic' [10].
(2) For similar properties of concurrent programs, one can oppose Owicki's non-compositional notion of 'interference freedom of proofs' [20], against Lamport's compositional 'Concurrent Hoare Logic' [12].
(3) For general temporal properties of concurrent programs, there is the work of Manna and Pnueli [16]. They reduce such properties to properties of so-called transitions - traces associated with atomic actions. To verify such transition properties, Floyd's inductive assertion method is extended. This contrasts with the current paper which proposes Transition Logic, supporting compositional reasoning about such properties.

The work of Z. Manna and A.Pnueli shows that the temporal behaviour of programs is provable in terms of transition properties. The motivation, for the present paper, to concentrate on transition properties goes deeper, as they are essential to obtaining compositionality. In the presence of concurrency, execution of a program is, in general, influenced by the (concurrent) actions of its environment, i.e., by the actions of the program(s) executing in parallel. Only for atomic, indivisibly executing, programs does it make sense to consider them in isolation from the environment.

This is obvious for the interleaving model of concurrency - adopted in this paper - in which atomic statements are executed one at the time. It is defendable for other models of concurrency, too, such as A. Salwicki's maximal parallelism [22] or the partial order semantics of W.Reisig [21], both grounded in Petri-net theory: If moves are in conflict, their execution is mutually exclusive. If they are conflict free, this intuitively means that execution of one move does not influence the execution of the others. In that case, one can view such moves as being executed in a strict but unspecified order; no change in the computed values will ensue.

The execution trace of a program within an environment is construed as being formed from the transitions of the program and the environment. The ordering of the transitions in the trace reflects the flow-of-control during execution. Consequently, in Transition Logic a program is viewed as offering a set of transitions which it can perform, subject to constraints imposed by the syntactic structure of the program, on the order in which the transitions can be taken; i.e., as a set of transitions constrained by the flow-of-control of the program.

Program specifications in Transition Logic take the form $[p]\alpha[q]$: Any transition of the program α that starts in a state satisfying the formula p, must end in a state satisfying q. To specify the flow-of-control, locations within programs will be labeled and these labels may appear in the formulae of the logic. Such labels also allow specifying the behaviour of each *individual* transition.

The next section introduces Transition Logic in more detail and defines the syntax and semantics of the logic w.r.t. a simple programming language - essentially while programs with parallel composition and variable sharing. Section 3 presents a formal system for this logic. Because validity depends on control locations too, the proof system contains an axiomatization of the flow-of-control in the programs discussed. The connection with temporal logic is described in section 4. Section 5 addresses issues of soundness and completeness. To render the rôle of the control-flow axiomatization explicit, first the basic properties on which completeness is based are formulated. Then, relative completeness of the logic is proved w.r.t. these properties. Finally, section 6 contains a conclusion and discusses some directions for further research.

This paper is clearly influenced by Lamport's work on invariance properties of concurrent programs, [12]. Lamport was the first to propose axiomatizing the flow-of-control, in the context of program verification. His 'Concurrent Hoare logic' was formalized by P. and R. Cousot in [5] . That paper contains a careful and abstract, language independent, analysis of the assumptions upon which completeness of the logic is based, that is of relevance to this paper, too. In [2], K. Apt and C. Delporte use similar ideas to prove certain eventuality properties of sequential programs.

The idea that control-flow forms an important aspect of concurrency is wellknown. That the formal study of flow-of-control is the fundamental principle behind compositional reasoning about properties of concurrent programs, seems to be less often realized.

2. TRANSITION LOGIC

Concurrency poses problems for compositional reasoning.

This is already apparent on a semantical level: Consider programs $\alpha \equiv x:=2$ and $\beta \equiv x:=1;x:=x+1$. For simplicity's sake, assume that the states of both programs consist of the single variable x, and let any execution start in the state satisfying $x=0$, i.e., in state 0. Denote the execution traces of α and β by $(0,2)$ and $(0,1,2)$, respectively. Now, consider the parallel execution of α and β: $\alpha \| \beta$. In any reasonable semantics, the moves of α and β will be interleaved. Hence, for $\alpha \| \beta$ the trace set $\{(0,2,1,2), (0,1,2,3), (0,1,2,2)\}$ is obtained. If the semantics of a program is defined as the set of its traces, then the principle of compositionality entails that the traces of $\alpha \| \beta$ should be obtainable from the traces of α and β. But how is one to construct, e.g., the trace $(0,1,2,3)$ from the traces $(0,2)$ and $(0,1,2)$? Such traces simply provide too little information, as state-changes caused by actions of the environment (β) of program α are ignored.

A similar observation can be made on the level of program specifications: The assertion $x \leq 2$ is an invariant property of the trace(s) of both α and β. This is expressed in a temporal logic like notation by: $\alpha : \Box(x \leq 2)$ and $\beta : \Box(x \leq 2)$. But although both α and β satisfy the same specification, $\alpha \| \beta : \Box(x \leq 2)$ is not valid, whereas $\alpha \| \alpha : \Box(x \leq 2)$ is. Again, too little information is provided to support compositionality.

The semantics of an atomic program can be defined as the set of its traces, because no concurrent action can change the program's behaviour. This suggests that the semantics of a non-atomic program may be cauched in terms of the sequences of atomic programs it can execute. More precisely: The semantics of a program will be a set consisting of sequences of traces; each such trace corresponds to the execution of an atomic action within the program. The ordering of the traces in a sequence, reflects the control-flow within the program. *As there is no information about the actions of the environment that may interleave, no relation will be assumed between the values of the program variables as recorded in the end-state of a trace and their values as recorded in the begin-state of the next trace in any trace sequence.*

As an example, the trace sequences of the programs α and β above, are $(a,2)$ and $(b,1)(c,c+1)$, where a, b and c denote arbitrary values. The set of trace sequences of $\alpha \| \beta$ is obtained, simply by interleaving the traces in the sequences of its components: $\{(a,2)(b,1)(c,c+1), (b,1)(a,2)(c,c+1), (b,1)(c,c+1)(a,2)\}$. Once it is decided that $\alpha \| \beta$ executes in an empty environment, the execution traces can be reconstructed: As no actions will interleave, in the second sequence necessarily $a=1$ and $c=2$. The corresponding trace is obtained by ignoring duplicate states; hence $(b,1,2,3)$. These ideas on compositional semantics for concurrency are not original; see, e.g., [1, 4] and for early references relating these ideas to the notion of resumptions, [3, 18].

Transition Logic concentrates on properties of trace sequences. This means that for any program, the logic must be able to specify the behaviour of its atomic programs and the (syntactic) constraints on the flow-of-control that the program imposes. For the latter, every statement in a program must be labeled and these labels can be used as propositions in the (state-) formulae of the logic. The program state is extended to provide truth-values for the program labels. Predictably, if a label, ℓ, is true in a state this means that in this state control within the program resides at the location labeled ℓ. A vital observation is that, whereas the environment can alter the values of the variables of a program, it cannot change the values of the labels in the program; i.e., it cannot modify the location at which control resides in the program. It is this fact that makes it possible to axiomatize the flow-of-control, necessary to obtain a complete proof system.

During execution of an atomic program, no actions interleave. Hence, two atomic programs are for all purposes equivalent, whenever they have the same input-output behaviour[1]. This is reflected in the basic formulae of Transition Logic, $[p]\alpha[q]$, in which atomic programs are viewed as transitions, transforming input-states into output-states: Any transition in α transforms a state satisfying p into a state satisfying q (remember that p and q may contain label-propositions).

One last point needs to be considered. If one wants to prove, say, $[p]\alpha;\beta[q]$ it would seem sufficient to show that $[p]\alpha[q]$ and $[p]\beta[q]$: The transitions of $\alpha;\beta$ are just the transitions of α plus the ones of β. For that matter, the same observation applies to a

---

[1] Having no output (-state) counts as behaviour, too. Of course, if one wants to distinguish between, e.g., failure and divergence, a special divergence (or failure) state is needed.

proof of $[p]\alpha\|\beta[q]$. Now a problem arises, because p and q will, in general, refer to labels in α and in β. However, in the transition sequences of α, nothing can be assumed about the truth-values of β's labels (and vice versa), as this entirely depends on the *context* in which α and β occur. In a context α;β, no label of β can be true when control remains within α; this is not so in a context $\alpha\|\beta$. These connections between the control locations of α and β are quite important. They form the reason that, e.g., the formula $[true]((x:=1;x:=2)\ell:)[\ell\rightarrow x=2]$ is a valid one, whereas a change of sequential into parallel composition in this program, invalidates it. Hence, Transition Logic allows formulae to appear in context: $\langle\beta|[p]\alpha[q]\rangle$, where β is a context in which α occurs. Such contexts enforce conditions on the label-valuations of states occurring (in this case) in the transition sequences of α. Such states must have label-valuations which are consistent with the syntactic structure of the context. I.e., a formula $\langle\alpha;\beta|in(\alpha)\rightarrow\neg in(\beta)\rangle$ will be valid, whereas the formula $\langle\alpha\|\beta|in(\alpha)\rightarrow\neg in(\beta)\rangle$ will not be valid (; in(α) expresses that control resides within α).

Introducing contexts solves the problem, but it is not claimed to be the only feasible solution. Another solution might be to disallow formulae $[p]\alpha[q]$, in which p or q contains labels that do not appear in α. There is a trade-off here, between simple proof rules and simple semantics. This paper opts for the simpler proof rules.

syntax.

Start with some 1st order similarity type t, which defines the predicate, function and constant symbols that appear in the programs and formulae, and a second similarity type l, which defines a countable set of 0-ary predicate letters (i.e., propositions) ℓ_0, ℓ_1, ..., disjoint from t, and functioning as labels.

Definition 2.0. The set of programs over t and l, PROG(tl), is inductively defined as the smallest set X such that

(1) $(\ell.x:=e.\ell')\in X$ and $(\ell.b?.\ell')\in X$ for all distinct labels ℓ and ℓ', variables x, and terms e and formulae b (atomic, if one wishes) over t

(2) $\alpha,\beta\in X \Rightarrow (\ell.\alpha\circ\beta.\ell')\in X$ where (a) ∘ is of the form ∪, ;, * or ∥, (b) no label in α (β) may appear in β (α), and (c) ℓ and ℓ' are distinct labels not appearing in α or β.

PROG(tl) is the set of regular programs with merge over assignments and tests. The program $\alpha*\beta$ is equivalent to $\alpha;(\beta;\alpha)^*$, and is introduced for technical reasons as explained in the next section. Note that $\alpha^*\equiv(true?*\alpha)$.

Definition 2.1. Transition Logic, TL(tl), is defined as the smallest set X such that

(1) $\langle\beta|p\rangle\in X$ for programs β and formulae p over tl

(2) $\langle\beta|[p]\alpha[q]\rangle\in X$ for programs β and subprograms α of β and formulae p and q over tl.

There is no constraint on the labels that appear in p and q. Some notation:

(1) LAB(α), LAB(p) - the set of labels appearing in program α respectively formula p,

(2) F(S) - the 1st order formulae over S,

(3) $\langle\beta|\alpha\rangle$ - α is a subprogram of β,

(4) $\langle\beta|\ell\|\alpha\rangle$ - ℓ occurs in a subprogram of β that can be executed parallel to α. Formally: There are programs $\bar{\alpha}$ and γ and labels m and m' such that $\langle\bar{\alpha}|\alpha\rangle$, $\ell\in LAB(\gamma)$ and either $\langle\beta|(m.\gamma\|\bar{\alpha}.m')\rangle$ or $\langle\beta|(m.\bar{\alpha}\|\gamma.m')\rangle$

(5) if $\alpha\equiv(\ell.\beta.\ell')$ then ${}^{\cdot}\alpha \stackrel{D}{=} \ell$, $\alpha^{\cdot} \stackrel{D}{=} \ell'$ and $\dot{\alpha} \stackrel{D}{=} (\vee\{\bar{\ell}\,|\,\bar{\ell}\in LAB(\alpha)\})\wedge\neg\ell'$.

In (5), ${}^{\cdot}\alpha$, $\dot{\alpha}$ and $\alpha^{\cdot}$, correspond to the location-predicates at(α), in(α) and after(α) of [12].

semantics.

A model for TL(tl) is a tuple, $\langle M,S\rangle$, where M is some t-structure (i.e., a model for F(t)) and S is the class of states over M; each state providing a valuation of the variables and of the label-propositions. Such models define

(1) a partial function $Trs: PROG(tl)^2\rightarrow P((S\times S)^\omega)$, where the range of Trs is the powerset of finite and infinite sequences of state-pairs, and

(2) a satisfaction relation $\models\subseteq\langle M,S\rangle\times TL(tl)$.

With any pair of programs, (β,α), such that $\langle\beta|\alpha\rangle$, Trs(β,α) associates the set of transition sequences obtained by executing α within the context β. First, an auxiliary function $Tr: PROG(tl)\rightarrow P((S\times S)^\omega)$ is defined, that associates transition sequences with programs in an empty environment.

The definition of Tr is standard, but for the label-valuations. The strategy in the inductive definition below will be to take states (for the induction base) and transition sequences (for the induction step) in which all labels whose truth-value will be affected, are set initially to false. Labels whose

value must change in a state, will be set explicitly.

There is one tricky point. If ℓ is some label that does not appear in a program α, nothing can be assumed about its truth-value during execution of α. Whether a transition in α will change its value or not, entirely depends on the context of α and ℓ, of which nothing is known. Compare, e.g., the contexts $\alpha;(\ell.\beta.\ell')$ and $\alpha^{\|}(\ell.\beta.\ell')$. This has the curious effect that any transition in a program must be allowed to change arbitrarily the truth-value of any label not appearing in the program.

In the definition below, state-variants are indicated as usual by $\sigma\{\cdot/\cdot\}$. This notation is (1) extended to denote variants of transitions: $\{\cdot/_1\cdot\}$, $\{\cdot/_2\cdot\}$ - denoting variants of the 1st and 2nd component of the state pair - and $\{\cdot/_{12}\cdot\}$ - denoting a variant of both components; and (2) elementwise extended to denote variants of sequences. Satisfaction of F(tl)-formulae, $M,\sigma\models p$, is assumed to be understood. To define the semantics of $(\ell.\alpha^*\beta.\ell')$, the syntax is (temporarily) extended to allow for any integer $n\geq 0$, programs $(\ell.\alpha(n)\beta.\ell')$. Abusing notation, we have $(\ell.\alpha^*\beta.\ell') \stackrel{D}{=} \cup\{(\ell.\alpha(n)\beta.\ell') \mid n\geq 0\}$.

From this definition, it follows that for any program α, the possible valuations of the labels appearing in α, only depend on the structure of α and (hence) do not change if α is embedded in some context $\bar{\alpha}$; provided α remains reachable in $\bar{\alpha}$. More precisely, the following *encapsulation* property holds:

For any programs α, $\bar{\alpha}$ and any label $\ell\in LAB(\alpha)$ such that $\langle\bar{\alpha}|\alpha\rangle$ and $\exists\bar{\tau}\in Tr(\bar{\alpha})\exists\bar{\sigma} in\bar{\tau}\ \bar{\sigma}(\cdot\alpha)=tt$:

$(\exists\tau\in Tr(\alpha)\exists\sigma in\tau\ \sigma(\ell)=w) \leftrightarrow (\exists\bar{\tau}\in Tr(\bar{\alpha})\exists\bar{\sigma} in\bar{\tau}\ \bar{\sigma}(\ell)=w)$,

where w denotes either tt or ff.

It is this property that makes the control-flow axiomatization in section 3 possible.

Next, to define Trs, the restrictions, imposed by a context (say, of α), on the values of the labels which do not occur in α, have to be enforced. This requires some additional machinery. A context β of a program α restricts the label valuations of α's transition sequences, so as to make them consistent with executing α in that context (β). Hence, it would seem that a trace $\tau\in Trs(\beta,\alpha)$ is obtained by "projecting" a trace $\bar{\tau}\in Tr(\beta)$ on the transitions in α. Such a trace, however, need not exist; consider, e.g., a context β=false?;α. The fact that α cannot be reached semantically, is of no concern for the consistency of α's label-valuations, which should depend on the syntactic structure of the context only. In fact, the straightforward axiomatization of the consistency of label-valuations in section 3 (LC_α) is rendered incomplete by taking semantic reachability into account. This suggests, defining for each program α, its "companion" α^c, obtained by replacing each test $(\ell.b?.\ell')$ in α by $(\ell.true?.\ell')$ (; in fact, it suffices to replace it by a test that can be passed from

Definition 2.3. For any $\alpha\in PROG(tl)$, inductively define $Tr(\alpha)$ by the following clauses.

(1) $\alpha=(\ell.b?.\ell')$ $\{(\sigma_1\{tt/\ell\},\sigma_2\{tt/\ell'\}) \mid \sigma_i\in S,\ \sigma_i(\ell)=\sigma_i(\ell')=ff\ i=1,2,\ \sigma_1|Var=\sigma_2|Var,\ M,\sigma_1'\models b\} = Tr(\alpha)$

(2) $\alpha=(\ell.x:=e.\ell')$ $\{(\sigma_1\{tt/\ell\},\sigma_2\{tt/\ell'\}) \mid \sigma_i\in S,\ \sigma_i(\ell)=\sigma_i(\ell')=ff\ i=1,2,\ \sigma_1\{\sigma_1(e)/x\}|Var=\sigma_2|Var\} = Tr(\alpha)$

For any $\alpha\in PROG(tl)$, $\tau_i\in Tr(\alpha_i)$ $i=1,2$ and labels ℓ, ℓ' such that $\alpha=(\ell.\alpha_1\circ\alpha_2.\ell')$ and for $i=1,2$ $len\tau_i=k_i$, $L_i=LAB(\alpha_i)$, $\tau_i[k_i]=(\sigma_i,\sigma_i')$, $\tau_i[1]=(\bar{\sigma}_i,\bar{\sigma}_i')$ and $\forall\bar{\ell}\in L_{i\,mod\,2+1}\ \tau_i(\ell)=\tau_i(\bar{\ell})=\tau_i(\ell')=ff$[1]:

(3) $\circ=\cup$ $\{\tau_i[1]\{tt/_1\ell\}\hat{}\tau_i[2:k_i-1]\hat{}\tau_i[k_i]\{tt/_2\ell'\} \mid i=1,2\} \subseteq Tr(\alpha)$

(4) $\circ=;$ $\tau_1[1]\{tt/_1\ell\}\hat{}\tau_1[2:k_1-1]\hat{}\tau_1[k_1]\{\bar{\sigma}_2(\bar{\ell})/_2\bar{\ell}\in L_2\}\hat{}\tau_2[1]\{\sigma_1'(\bar{\ell})/_1\bar{\ell}\in L_1\}\hat{}\tau_2[2:k_2-1]\hat{}\tau_2[k_2]\{tt/_2\ell'\} \in Tr(\alpha)$

(5) $\circ=(0)$ $\tau_1[1]\{tt/_1\ell\}\hat{}\tau_1[2:k_1-1]\hat{}\tau_1[k_1]\{tt/_2\ell'\} \in Tr(\alpha)$

(6) $\circ=(n+1)$ $\{\tau[1:k-1]\hat{}\tau[k]\{ff,\bar{\sigma}_2(\bar{\ell})/_2\ell',\bar{\ell}\in L_2\}\hat{}\tau_2[1]\{\sigma'(\bar{\ell})/_1\bar{\ell}\in L_1\}\hat{}\tau_2[2:k_2-1]\hat{}\tau_2[k_2]\{tt,\bar{\sigma}_1(\bar{\ell})/_2\ell,\bar{\ell}\in L_1\}\hat{}\tau_1[1]\{tt,\sigma_2'(\bar{\ell})/_1\ell,\bar{\ell}\in L_2\}\hat{}\tau_1[2:k_1-1]\hat{}\tau_1[k_1]\{tt/\ell'\} \mid \tau\in Tr((\ell.\alpha_1(n)\alpha_2.\ell')),\ len\tau=k,\ \tau[k]=(\sigma,\sigma')\} \subseteq Tr(\alpha)$

(7) $\circ=*$ $Tr((\ell.\alpha_1(n)\alpha_2.\ell')) \subseteq Tr(\alpha)$ for all $n\geq 0$

(8) $\circ=\|$ $\{\tau[1]\{tt/_1\ell\}\hat{}\tau[2:k-1]\hat{}\tau[k]\{tt/_2\ell'\} \mid \tau\in merge(\bar{\tau}_1,L_1,\bar{\tau}_2,L_2),\ k=len\tau,$ for $i=1,2$ $\bar{\tau}_{i\,mod\,2+1}=\tau_{i\,mod\,2+1}\{\bar{\sigma}_i(\bar{\ell})/_{12}\bar{\ell}\in L_i\}\} \subseteq Tr(\alpha)$

where $merge(\tau,\Gamma,\tau',\Gamma') \stackrel{D}{=} lmerge(\tau,\Gamma,\tau',\Gamma') \cup lmerge(\tau',\Gamma',\tau,\Gamma)$

and $lmerge(\tau,\Gamma,\tau',\Gamma') \stackrel{D}{=} \begin{cases}\{\tau'\} & \text{if } len=0,\\ \{\tau[1]\hat{}\bar{\tau} \mid \bar{\tau}\in merge(\tau[2:],\Gamma,\tau'\{\bar{\sigma}'(\ell)/_{12}\ell\in\Gamma\},\Gamma'),\ \tau[1]=(\bar{\sigma},\bar{\sigma}')\} & \text{otherwise}\end{cases}$

[1] The value of $\ell,\bar{\ell}$ and ℓ' in any state of τ_i is false.

at least one state). In α^c, any execution path that is syntactically possible (in α), is semantically possible.

Now, auxiliary predicates, $\beta\text{-sg} \subseteq S$ (called β-state good) and $\beta\text{-tg} \subseteq S^2$ (called β-transition good) can be defined, to express that a state or a transition has valuation(s) that are consistent with β's syntax:

Definition 2.4. For any $\beta \in PROG(tl)$, $\sigma \in S$, $L \stackrel{D}{=} LAB(\beta)$

(1) $\beta\text{-sg}(\sigma)$ iff $\exists \tau \in Tr(\beta^c) \exists i \geq 1\ \tau[i] = (\bar{\sigma}, \bar{\sigma}') \Rightarrow \bar{\sigma}|L = \sigma|L \vee \bar{\sigma}'|L = \sigma|L$

(2) $\beta\text{-tg}(\sigma,\sigma')$ iff $\exists \tau \in Tr(\beta^c) \exists i \geq 1\ \tau[i] = (\bar{\sigma}, \bar{\sigma}') \Rightarrow \bar{\sigma}|L = \sigma|L \wedge \bar{\sigma}'|L = \sigma'|L$

Note the disjunction in (1) and the conjunction in (2).

Definition 2.5. For any $\alpha, \beta \in PROG(tl)$ such that $\langle \beta | \alpha \rangle$ and $\tau \in (S \times S)^\omega$:

$\tau \in Trs(\beta,\alpha)$ iff $\tau \in Tr(\alpha) \wedge \forall i (\tau[i] = (\sigma,\sigma') \Rightarrow \beta\text{-tg}(\sigma,\sigma'))$

Finally, validity and satisfaction in a model can be defined:

Definition 2.6. Let $p,q \in F(tl)$, $\Phi \in TL(tl)$, $\alpha, \beta \in PROG(tl)$ and $\langle \beta | \alpha \rangle$. Then

(1) $\models \Phi$ iff $\forall$ models $\langle M,S \rangle$ of $TL(tl)$ $\langle M,S \rangle \models \Phi$

(2) $\langle M,S \rangle \models \langle \beta | p \rangle$ iff $\forall \sigma \in S\ \beta\text{-sg}(\sigma) \Rightarrow M,\sigma \models p$

(3) $\langle M,S \rangle \models \langle \beta | [p]\alpha[q] \rangle$ iff $\forall \tau \in Tr(\beta,\alpha) \forall i \geq 1$ $\tau[i] = (\sigma,\sigma') \Rightarrow (M,\sigma \models p \Rightarrow M,\sigma' \models q)$

3. THE PROOF SYSTEM, PS(M).

As notation for proof rules, $\langle \beta | A_1, \ldots, A_n \Rightarrow C \rangle$ is used, with the usual interpretation: To infer $\langle \beta | C \rangle$, prove $\langle \beta | A_1 \rangle$, ..., $\langle \beta | A_n \rangle$.

The system consists of three parts. The first part concerns the composition of program (-proofs) and consists of the 4 Composition rules, the Test and the Assignment rule :

For o of the form $\cup$, ;, * or $\|$, C_o.
$\langle \beta | [p]\alpha_1[q], [p]\alpha_2[q] \Rightarrow [p](\ell.\alpha_1 o \alpha_2.\ell')[q] \rangle$

T. $\langle \beta | (p \wedge b \wedge \alpha^\cdot \rightarrow q) \Rightarrow [p]\alpha[q] \rangle$ provided $\alpha \equiv (\ell.b?.\ell')$ and $LAB(p) = \emptyset$

As. $\langle \beta | (\alpha^\cdot \wedge \exists y(p[y/x] \wedge x = e[y/x]) \rightarrow q) \Rightarrow [p]\alpha[q] \rangle$ provided $\alpha \equiv (\ell.x := e.\ell')$, $LAB(p) = \emptyset$ and $y \notin Fvar(p,e) \cup \{x\}$

Next, follow some auxiliary rules and axioms. The Invariance, Strengthen and the 2 Extension rules, the Oracle rule and the Propositional tautology axiom:

Inv. $\langle \beta | p \Rightarrow [true]\alpha[p] \rangle$

S. $\langle \beta | [p \wedge q]\alpha[r], \alpha^\cdot \rightarrow q, [q]\alpha[q \vee \alpha^\cdot] \Rightarrow [p]\alpha[r] \rangle$, $q \in F(1)$

E_1. prove $\langle \beta | p \rangle$ to infer $\langle \beta' | p \rangle$ provided $\langle \beta' | \beta \rangle$ and $p \in F(1)$

E_2. prove $\langle \beta | [p]\alpha[q] \rangle$ to infer $\langle \beta' | [p]\alpha[q] \rangle$ provided $\langle \beta' | \beta \rangle$

O. $\langle \beta | p \rangle$ for any $p \in F(tl)$ such that $M \models p$

Pt. All instances of propositional tautologies

Finally, the control flow axioms (Flow and Label Consistency). The formulae $LC(\alpha)$ are defined below:

$F_{1,2}$. $\langle \beta | [\neg\ell]\alpha[\neg\ell] \rangle$ provided $\langle \beta | \ell \| \alpha \rangle$ and α is atomic

LC_α. $\langle \alpha | LC(\alpha) \rangle$ for any $\alpha \in PROG(tl)$

The recursively enumerable set of label consistency axioms, LC_α, axiomatizes the consistency of label valuations. The $F(1)$-formulae $LC(\alpha)$ are recursively defined as follows ($\bar{\vee}$ denotes the exclusive or)

(1) $\alpha \equiv (\ell.b?.\ell')$ or $(\ell.x := e.\ell') \Rightarrow LC(\alpha) \stackrel{D}{=} \ell\ \bar{\vee}\ \ell'$
if $\alpha \equiv (\ell.\alpha_1 o \alpha_2.\ell')$ then $LC(\alpha) \stackrel{D}{=} LC(\alpha_1) \wedge LC(\alpha_2) \wedge C$, where C is defined by the following clauses:

(2) $o \equiv \cup \Rightarrow \ell \leftrightarrow (\dot{}\alpha_1 \bar{\vee} \dot{}\alpha_2) \wedge \ell' \leftrightarrow (\alpha_1' \bar{\vee} \alpha_2') \wedge \neg((\dot{\alpha}_1 \vee \alpha_1') \wedge (\dot{\alpha}_2 \vee \alpha_2'))$

(3) $o \equiv ; \Rightarrow \ell \leftrightarrow \dot{}\alpha_1 \wedge \alpha_1' \leftrightarrow \dot{}\alpha_2 \wedge \alpha_2' \leftrightarrow \ell' \wedge \neg(\dot{\alpha}_1 \wedge (\dot{\alpha}_2 \vee \alpha_2'))$

(4) $o \equiv * \Rightarrow \ell \leftrightarrow \dot{}\alpha_1 \wedge \alpha_1' \leftrightarrow (\dot{}\alpha_2 \bar{\vee} \ell') \wedge \alpha_2' \rightarrow \ell \wedge \neg(\dot{\alpha}_1 \wedge \dot{\alpha}_2)$

(5) $o \equiv \| \Rightarrow \ell \leftrightarrow (\dot{}\alpha_1 \wedge \dot{}\alpha_2) \wedge \ell' \leftrightarrow (\alpha_1' \wedge \alpha_2') \wedge (\dot{\alpha}_1 \vee \alpha_1' \leftrightarrow \dot{\alpha}_2 \vee \alpha_2')$

$LC(\alpha)$ expresses that a valuation of the labels of α is consistent with α's syntactic structure. Such a recursive definition is possible, because this consistency does not depend on α's environment, i.e., because the encapsulation property holds. This would have not been true, had the usual Kleene-* been used instead of the binary *: $LC((\ell.\alpha^*.\ell'))$ necessarily will allow the inference $\alpha^\cdot \rightarrow \dot{}\alpha$ (or $\dot{}\alpha \rightarrow \alpha^\cdot$) in order to model looping. As $\dot{}\alpha, \alpha^\cdot \in LAB(\alpha)$, this means that now, a context of α can induce new relations between α's labels.

Some useful derived rules and axioms are:

I. $\langle \beta | p \rightarrow p', [p']\alpha[q'], q' \rightarrow q \Rightarrow [p]\alpha[q] \rangle$

$C_{\wedge,\vee}$. $\langle \beta | [p]\alpha[q], [p']\alpha[q'] \Rightarrow [p \stackrel{\wedge}{\vee} p']\alpha[q \stackrel{\wedge}{\vee} q'] \rangle$

F_3. $\langle \beta | [true]\alpha[\alpha^\cdot] \rangle$ provided α is atomic

F_4. $\langle \beta | [true]\alpha[\dot{\alpha} \vee \alpha^\cdot] \rangle$

Axiom F_3 easily follows from T and As. Proofs for the other axiom and the proof rules are constructed using structural induction w.r.t. α and, for F_4, applying the trivial observation that $\models \dot{\gamma} \vee \gamma \leftrightarrow \dot{\alpha} \vee \alpha^\cdot$ if $\langle \alpha | \gamma \rangle$ (cf. the definition of $\dot{\alpha}$ and $\alpha^\cdot$).

example.

Consider Lamport's Concurrent Hoare logic [12]. His sequential composition rule is a derived rule of PS(M). Lamport deals with safety properties and his

specifications, $\{p\}\alpha\{q\}$, have the interpretation: "If execution is begun anywhere in α with the predicate p true, then executing α will leave p true while control is inside α, and will make q true if and when α terminates". Consequently, $\{p\}\alpha\{q\}$ translates into Transition Logic as: $[p]\alpha[\dot{\alpha}\rightarrow p\wedge\alpha'\rightarrow q]$ (in [10] the context is always left implicit).

In this paper's notation, Lamport's sequential composition rule becomes:

SC. $\langle\delta\,|\,[t]\alpha[\dot{\alpha}\rightarrow t\wedge\alpha'\rightarrow u]\,,\ [v]\beta[\beta\rightarrow v\wedge\beta'\rightarrow w]\,,\ u\wedge'\beta\rightarrow v\ \Rightarrow$
$[\dot{\alpha}\rightarrow t\wedge\beta\rightarrow v]\gamma[\dot{\alpha}\rightarrow t\wedge\beta\rightarrow v\wedge\gamma'\rightarrow w]\rangle$,
where $\gamma\equiv(\ell.\alpha;\beta.\ell')$ and $\langle\delta\,|\,\gamma\rangle$.

To derive SC, it will be convenient to generalize it to:

SC'. $\langle\delta\,|\,[p]\alpha[q]\,,\ [r]\beta[s]\Rightarrow[A]\gamma[C]\rangle$, where γ and δ are as in SC, $A\equiv\dot{\alpha}\rightarrow p\wedge\beta\rightarrow r$ and $C\equiv(\dot{\alpha}v\alpha'\rightarrow q)\wedge((\neg'\beta\wedge(\beta v\beta'))\rightarrow s)$

It is not difficult to see that SC' is sound. To derive SC' requires some more effort:

We want to use the $C_;$ rule. So, first derive $\vdash\langle\delta\,|\,[A]\alpha[C]\rangle$. In the derivation, the context is suppressed.

$$\dfrac{\dfrac{\dfrac{[\text{true}]\alpha[\dot{\alpha}v\alpha']}{\dot{\alpha}\rightarrow\text{true}}C\wedge\quad\dfrac{[\dot{\alpha}]\alpha[\dot{\alpha}v\alpha']\qquad[p]\alpha[q]}{[\dot{\alpha}\wedge p]\alpha[(\dot{\alpha}v\alpha')\wedge q]}C\wedge}{\dot{\alpha}\wedge A\rightarrow\dot{\alpha}\wedge p}\qquad\dfrac{LC(\gamma)\rightarrow(\dot{\alpha}v\alpha'\rightarrow('\beta v\neg'(\beta v\beta')))}{(\dot{\alpha}v\alpha')\rightarrow('\beta v\neg'(\beta v\beta'))}Pt\quad\dfrac{LC(\gamma)}{C}Pt}{[\dot{\alpha}\wedge A]\alpha[C]}I$$

$$\dfrac{'\alpha\rightarrow\dot{\alpha}\qquad[\text{true}]\alpha[\dot{\alpha}v\alpha']}{[A]\alpha[C]}S$$

The proof of $\langle\delta\,|\,[A]\beta[C]\rangle$ is completely analogous. Now, SC' is obtained by a final application of $C_;$.

It remains to show that SC is a derived rule, too. To prove this, apply SC' with $p\equiv t$, $q\equiv(\dot{\alpha}\rightarrow t\wedge\alpha'\rightarrow u)$, $r\equiv v$ and $s\equiv\beta\rightarrow v\wedge\beta'\rightarrow w$). Formula A has the required form, C not yet. Observe that
$\models C\leftrightarrow(\dot{\alpha}\rightarrow t\ \wedge\ \alpha'\rightarrow u\ \wedge\ \neg'\beta\wedge\beta\rightarrow v\ \wedge\ \neg'\beta\wedge\beta'\rightarrow w)$ and that
$\models LC(\delta)\rightarrow(\alpha'\leftrightarrow'\beta\ \wedge\ \beta'\leftrightarrow\gamma'\ \wedge\ \beta'\rightarrow\neg'\beta)$.
Hence, using O, Pt and LC_δ, we obtain
$\{\langle\delta\,|\,[A]\gamma[C]\rangle\}\vdash\langle\delta\,|\,[A]\gamma[\dot{\alpha}\rightarrow t\ \wedge\ '\beta\rightarrow u\ \wedge\ \neg'\beta\wedge\beta\rightarrow v\ \wedge\ \beta'\rightarrow w]\rangle$.
To obtain the consequent of SC, use the assumption $\langle\delta\,|\,u\wedge'\beta\rightarrow v\rangle$, to replace $'\beta\rightarrow u$ by $'\beta\rightarrow v$.

4. THE CONNECTION WITH TEMPORAL LOGIC.

Transition Logic cannot reason about and specify temporal properties in a direct way, like the logics of Lamport [13] and Manna and Pnueli [16] can. This ability was sacrificed to obtain compositionality. However, this does not imply a loss in reasoning power. To wit: Any temporal property, expressible in Manna and Pnueli's Temporal Program Logic (TPL), that is true of a program α ($\alpha\in PROG(tl)$), can be proved hence reasoned about in a suitable extension of Transition Logic.

This claim follows from a straightforward translation of results from [15, 16, 17]: TPL reduces proofs of temporal properties to proofs of classical properties without temporal modalities. These classical properties are readily translated into Transition Logic formulae. In [17], rules are introduced for certain classes of temporal formulae, which are complete in the sense that to prove a formula from any of these classes, one application of the respective rule suffices, the rest is classical reasoning (which can be done in PS(tl)). The claim follows by extending PS(tl) with these rules and by observing that (for programs in PROG(tl)) any temporal formula can be reduced to a formula in one of the above classes.

Note that no justness anf fairness constraints are imposed upon the executions of PROG(tl)-programs. Hence, although TPL is capable of dealing with such constraints, this aspect is ignored here. The rest of the section only states the facts and (hence) the reader should be familiar with [15, 16, 17].

In TPL, (temporal) formulae are interpreted over a fixed set of state-sequences; that is, over the execution traces of a fixed program. Consequently, there is no notion of composition of programs in TPL. Hence, with each program $\alpha\in PROG(tl)$, associate its set of traces, $Et(\alpha)$ (obtained from $Tr(\alpha)$). As in TPL, if ϵ is a trace of α ($\epsilon\in Et(\alpha)$), then by definition any non-empty suffix of ϵ is a trace of α, too.

Temporal logic, TeL(tl), is obtained by extending classical 1st order logic F(tl), with 3 temporal modalities: $\bigcirc$ (next state), $\Diamond$ (eventuality) and $\mathcal{U}$ (weak until or unless). The formulae of TeL(tl) are defined as usual, by inductively applying these modalities, starting with F(tl)-formulae. It is a fact that any temporal modality can be expressed using

○, ◇ and U. The interpretation of Tel(tl) (over Et(α)) is inductively defined as follows:

$\langle M,S\rangle\models\alpha:\Phi$ iff $\forall\varepsilon\in Et(\alpha)\ M,\varepsilon\models\Phi$

$M,\varepsilon\models p$ iff $M,\varepsilon[1]\models p$ (for $p\in F(tl)$)

$M,\varepsilon\models\bigcirc\Phi$ iff $len\varepsilon\geq 1$ and $M,\varepsilon[2:]\models\Phi$

$M,\varepsilon\models\Phi\mathcal{U}\Psi$ iff $\exists i\leq len\varepsilon\ M,\varepsilon[i:]\models\Phi$

$M,\varepsilon\models\Phi\mathcal{U}\Psi$ iff $\forall i\leq len\varepsilon\ M,\varepsilon[i:]\models\Phi$ or $\exists j\leq len\varepsilon\ \forall 1\leq i<j\ M,\varepsilon[i:]\models\Phi\wedge M,\varepsilon[j:]\models\Psi$

In [17] three rules are introduced for proving properties of the form $p\rightarrow\Diamond q$, $p\rightarrow\bigcirc q$ and $p\rightarrow q\mathcal{U}r$, where p, q and r are state-formulae, i.e., $p,q,r\in F(tl)$. Provided $F(tl)$ meets certain requirements, these rules are shown to be complete in the above sense that any of these properties can be proved, using one single application of the respective rule as the only temporal step. I.e., no temporal modalities appear in the premisses of these rules. In this paper's terminology, the rules are:

$$\text{N.}\ \frac{[p\wedge\bar{p}]\alpha[q]\ ,\ p\rightarrow\neg\dot{\alpha}\qquad \dot{\alpha}\rightarrow\bar{p}\ ,\ [\bar{p}]\alpha[\bar{p}]}{\alpha:p\rightarrow\bigcirc q}$$

$$\text{U.}\ \frac{[\bar{q}\wedge\bar{p}]\alpha[\bar{r}\vee\bar{q}]\ ,\ \dot{\alpha}\rightarrow\bar{p}\ ,\ [\bar{p}]\alpha[\bar{p}]\qquad p\wedge\bar{p}\rightarrow\bar{q}\vee\bar{r}\ ,\ \bar{q}\wedge\bar{p}\rightarrow q\ ,\ \bar{r}\wedge\bar{p}\rightarrow r}{\alpha:p\rightarrow q\mathcal{U}r}$$

$$\text{E.}\ \frac{[\bar{q}(n)]\alpha[q\vee(\dot{\alpha}\wedge\exists m<n.\bar{q}(m))]\qquad \dot{\alpha}\rightarrow\bar{p}\ ,\ [\bar{p}]\alpha[\bar{p}]\ ,\ p\wedge\bar{p}\rightarrow q\vee(\dot{\alpha}\wedge\exists n.\bar{q}(n))}{\alpha:p\rightarrow\Diamond q}$$

where n and m are natural numbers.

In these rules, the TL-formula $[p]\alpha[q]$ translates the TPL-expression "α leads from p to q", with interpretation [17]: For any atomic action τ of α and for any states s and s', $p(s)\wedge s'\in M[\![\tau]\!]s\rightarrow q(s')$. Note that as a consequence, PS(tl) allows formal reasoning about such TPL-expressions; something that is missing in [16, 17].

Rules E and N have been changed w.r.t the original rules of [17]: In TPL, traces of programs are always infinite (if necessary, the terminal state of such a trace is repeated); not so for the traces in Et(α). This accounts for the additional premiss $p\rightarrow\neg\dot{\alpha}$ in N and for the appearance of $\dot{\alpha}$ in the first and last premiss of E. Moreover, the E-rule could be simplified, because justness and fairness issues are ignored.

The proofs in [15, 17] indicate that, in order to obtain completeness of these rules, the following predicates and relation must be F(tl)-definable:
(1) "$\bar{\tau}$ is a finite prefix of some $\tau\in Et(\alpha)$",
(2) "$M,\sigma\models p$" (for $p\in F(tl)$),
(3) "n is a standard natural number" and "$n<m$".

Rule E uses a well-foundedness argument to show eventuality. This explains the third requirement. Also, the formula $\bar{q}(n)$ in rule E, will have in general to express something like "α will establish q in at most n more execution steps". Hence, requirements 1 and 2.

Now, what about the proof of general temporal properties? Observe that no transition sequence in $Tr(\alpha)$, hence no trace in $Et(\alpha)$, has infinite length. This implies that "$\tau\in Et(\alpha)$" is F(tl)-definable, too. Consequently, "$M,\tau\models\Phi$" is definable for any $\Phi\in Tel(tl)$ and $\tau\in Et(\alpha)$, so that any temporal formula can be reduced to one of the standard forms $p\rightarrow\bigcirc q$, $p\rightarrow\Diamond q$ or $p\rightarrow q\mathcal{U}r$.

We agree that the use of such coding tricks is unsatisfactory but observe that such tricks have to be used in TPL, too.

As an example, let p_e be the signature of Peano-arithmetic, extended with a unary predicate (symbol) nat. Let M be a class of models in which nat is interpreted as the defining predicate of the standard natural numbers (and in which the symbols in p receive their usual interpretation). Clearly, $F(p_e1)$ satisfies the definability requirements 1, 2 and 3. Hence, assuming for the moment that $PS(M)$ is complete (w.r.t. M), then

$$M\models\alpha:\Phi\ \Leftrightarrow\ PS(M)\cup\{N,U,E\}^{1}\vdash\Phi,\ \alpha\in PROG(p_e1),\ \Phi\in TeL(p_e1).$$

This can of course be generalized to any signature that allows an arithmetical model (cf. Harel's [9]).

5. SOUNDNESS AND COMPLETENESS.

PS(M) is an ordinary Hilbert-style proof system. Consequently, to establish soundness, it suffices to show that each individual proof rule is sound and that every axiom scheme is valid.

In this section, references to models, M, are usually suppressed. Also, if ℓ is a label then $\tilde{\ell}\in F(l)$ denotes uniformly either the formula ℓ or $\neg\ell$, in expressions.

Lemma 5.0. E_1, 0, Pt, $F_{1,2}$ and LC_α are valid.

Proof.

Rule E_1. Remember that $\langle M,S\rangle\models\langle\beta'|p\rangle\ \Leftrightarrow\ (\forall\sigma\in S\ \beta'\text{-sg}(\sigma)\Rightarrow M,\sigma\models p)$. Now, observe that $\beta'\text{-sg}\subseteq\beta\text{-sg}$. This is a consequence of the encapsulation property and the fact that for $\ell\notin LAB(\beta)$, β-sg imposes no con-

[1] Strictly speaking, an additional rule is needed, allowing the substitution of equivalent formulae in temporal formulae, so as to be able to reduce to standard form.

straint on their values.

O,Pt. Trivial and uninteresting.

$F_{1,2}$. By definition of $\langle\beta|\ell\|\alpha\rangle$, there is a subprogram of β, $\rho\equiv(m.\gamma\|\delta.m')$ such that $\langle\gamma|\alpha\rangle$ and $\ell\in LAB(\delta)$ or vice versa. W.l.o.g., assume that $\langle\gamma|\alpha\rangle$. The encapsulation property implies that it suffices to prove that for any transition (σ,σ') caused by an action in γ, $\rho\text{-}tg(\sigma,\sigma')=\sigma(\ell)=\sigma'(\ell)$. This follows directly from the definition of merge.

LC_α. This is proved using structural induction. If α is atomic, LC_α is easily seen to hold. The induction step amounts to a depressing analysis of how composite transition sequences are constructed and is left to the reader.

Lemma 5.1. C_o, T, As, Inv, S and E_2 are sound rules.

Proof.

As. Choose any $(\sigma,\sigma')\in Trs(\beta,\alpha)$ such that $\sigma\models p$. To show: $\sigma'\models q$.
Assume that $\sigma'\not\models\alpha^*\wedge\exists y(\ldots)$. As $\sigma'\models\alpha^*$, this implies $\sigma'\{\nu/y\}\not\models(\ldots)$ for every $\nu\in|M|$; in particular $\sigma'\{\bar\nu/y\}\not\models(\ldots)$, where $\bar\nu=\sigma(x)$. Because $\sigma'\{\bar\nu/y\}\models x=e[y/x]$ $(y\notin Fvar(e)\cup\{x\})$, also $\sigma'\{\bar\nu/y\}\not\models p[y/x]$. Consequently, $\sigma'\{\bar\nu/x\}\not\models p$ $(y\notin Fvar(p))$. As $LAB(p)=\emptyset$, this implies $\sigma\not\models p$; a contradiction. Hence $\sigma'\models\alpha^*\wedge\exists y(\ldots)$, so that $\sigma'\models q$.

C_o, T, Inv, S, E_2. These rules are even more trivial to prove sound and are left as exercises.

Theorem 5.0. $PS(M)\vdash\phi\ \Rightarrow\ M\models\phi$ for any $\phi\in Tl(tl)$.

Proof. An easy induction w.r.t. the complexity of the proof.

completeness.
PS(M) is complete relative to the theory of M (cf. the Oracle axiom). Notably, no further restrictions on models M are needed, such as Cook's notion of expressiveness. The completeness proof does rely on the definability of the strongest post condition (spc) for atomic programs. However, in TL this is definable in any (1st order) model. Of course, section 4 indicated that in order to prove interesting properties, models do have to meet additional criteria.

The following two properties are essential for obtaining completeness:

(A) $LC(\beta)$ defines β-sg: $\forall\sigma\in S\ M,\sigma\models LC(\beta)\ \Leftrightarrow\ \beta\text{-}sg(\sigma)$

(B) Let $\alpha,\beta\in PROG(tl)$, α atomic, $\langle\beta|\alpha\rangle$ and $\ell\in LAB(\beta)$.
If $\langle\beta|\ell\|\alpha\rangle$ then the value of ℓ remains unaffected by executing α : $\forall(\sigma,\sigma')\in Trs(\beta,\alpha)\ \sigma(\ell)=\sigma'(\ell)$.
If $\neg\langle\beta|\ell\|\alpha\rangle$ then the value of ℓ in a state produced by α is independent of the state in which α was executed: For w either tt or ff,
$(\exists(\sigma,\sigma')\in Trs(\beta,\alpha)\forall\sigma''\ (\sigma,\sigma'')\in Trs(\beta,\alpha)\Rightarrow\sigma''(\ell)=w)\ \Rightarrow$
$\forall(\bar\sigma,\bar\sigma')\in Trs(\beta,\alpha)\ \bar\sigma'(\ell)=w$

The proofs of these properties are straightforward but very lengthy analyses of the construction of the label valuations in the definition of Tr and will not be given in this paper.

The completeness proof splits into a number of cases. The easiest case is the subject of

Theorem 5.1. $M\models\langle\beta|p\rangle\ \Rightarrow\ PS(M)\vdash\langle\beta|p\rangle$ $(p\in F(tl))$.

Proof. Property (A) implies that $M\models LC(\beta)\rightarrow p$. Now, use O, LC_β and simple propositional reasoning to obtain $\vdash\langle\beta|p\rangle$.

Completeness for transition formulae is based on the definability of spc. Hence:

Lemma 5.2. Let $\alpha,\beta\in PROG(tl)$, α atomic, $\langle\beta|\alpha\rangle$ and $p\in F(tl)$. Then the spc of p w.r.t. $\langle\beta|\alpha\rangle$, $sp(p,\langle\beta|\alpha\rangle)$, is $F(tl)$-definable.

Proof. As usual $M,\bar\sigma\models sp(p,\langle\beta|\alpha\rangle)\ \Leftrightarrow\ \exists(\sigma,\sigma')\in Trs(\beta,\alpha)$ $M,\sigma\models p\&\sigma'=\bar\sigma$. Define $L\stackrel{D}{=}\wedge\{\ell\vee\neg\ell\,|\,\ell\in LAB(p)\}$ and bring L into disjunctive normal form[1] $L_1\vee\ldots\vee L_n$. Each L_i is a conjunction of atomic formulae of the form ℓ or $\neg\ell$ and $LAB(L_i)=LAB(p)$. Obtain assertions $p_i\in F(t)$ by substituting in p, true or false for any $\ell\in LAB(p)$ depending on whether ℓ or $\neg\ell$ appears in L_i. By definition $LAB(p_i)=\emptyset$. Clearly, $\models\langle\beta|L_i\rightarrow(p\leftrightarrow p_i)\rangle$ hence $\models\langle\beta|p\leftrightarrow\vee\{p_i\wedge L_i\,|\,i=1..n\}\rangle$. Now, for any p_i, let $\bar p_i$ denote the formula $p_i\wedge b$ in case $\alpha\equiv(\ell.b?.\ell')$ and the formula $\exists y(p_i[y/x]\wedge x=e[y/x])$ in case $\alpha\equiv(\ell.x:=e.\ell')$ $(y\notin Fvar(p,e)\cup\{x\})$.
Let for any L_i, $\bar L_i$ denote the F(1)-formula $\wedge\{\ell\,|\models\langle\beta|[L_i]\alpha[\ell]\rangle\}\wedge\wedge\{\neg\ell\,|\models\langle\beta|[L_i]\alpha[\neg\ell]\rangle\}$. Then it is a simple exercise to show that
$sp(p,\langle\beta|\alpha\rangle)\stackrel{D}{=}\alpha^*\wedge LC(\beta)\wedge\vee\{\bar L_i\wedge\bar p_i\,|i=1..n\}$.
This representation will be used in the sequel.

Lemma 5.3. $PS(M)\vdash\langle\beta|[L]\alpha[sp(L,\langle\beta|\alpha\rangle)]\rangle$, where α is atomic and $L\in F(1)$.

Proof. Let $\bar L$ denote the spc. There are two cases.

---

[1] Strictly speaking, true is a d.n.f. of L, too. What is meant, here and in the sequel, is to apply the well-known algorithm which (syntactically) transforms a formula into normal form.

(1) $\models\langle\beta|\alpha^{\cdot}\rightarrow\neg L\rangle$: In the following derivation, the context is suppressed.

$$\dfrac{\dfrac{[\mathrm{false}]\alpha[\bar{L}] \qquad L\wedge\neg L\rightarrow\mathrm{false}}{[L\wedge\neg L]\alpha[\bar{L}]}\,I \qquad \alpha^{\cdot}\rightarrow\neg L \qquad \dfrac{[\mathrm{true}]\alpha[\alpha^{\cdot}] \qquad \alpha^{\cdot}\rightarrow\alpha^{\cdot}\vee L}{[\mathrm{true}]\alpha[\alpha^{\cdot}\vee L]}\,I}{[L]\alpha[\bar{L}]}\,S$$

(2) $\not\models\langle\beta|\alpha^{\cdot}\rightarrow\neg L\rangle$: If $\models\langle\beta|\bar{L}\rightarrow\mathrm{false}\rangle$, then α has no output-state. This is only possible if $\alpha\equiv(\ell.\mathrm{false}?.\ell')$. But then, T can be used to obtain $\vdash\langle\beta|[L]\alpha[\mathrm{false}]\rangle$. Hence, assume $\not\models\langle\beta|\bar{L}\rightarrow\mathrm{false}\rangle$. This means that there exists at least one transition $(\sigma,\sigma')\in Trs(\beta,\alpha)$ such that $\sigma'\models L$. Consequently, property (B) implies the following facts:

(1) if $\langle\beta|\ell\|\alpha\rangle$ then $\models\langle\beta|[L]\alpha[\ell]\rangle\Rightarrow\models\langle\beta|[\ell]\alpha[\ell]\rangle$, and

(2) if $\neg\langle\beta|\ell\|\alpha\rangle$ then $\models\langle\beta|[L]\alpha[\ell]\rangle\Rightarrow\models\langle\beta|\alpha^{\cdot}\rightarrow\ell\rangle$.

We intend to use the representation of $\bar{L}$ as in the proof of lemma 5.2. So, bring L into disjunctive normal form $L_1\vee\ldots\vee L_n$ and let S denote either the formula b or $\exists y.x=e[y/x]$[1] depending on whether α is a test or an assignment. Then, using the notation of lemma 5.2: $\bar{L}\leftrightarrow S\wedge\alpha^{\cdot}\wedge LC(\beta)\wedge(\bar{L}_1\vee\ldots\vee\bar{L}_n)$. Now, it is easy to obtain $\vdash\langle\beta|[L]\alpha[S\wedge LC(\beta)]\rangle$. Facts (1) and (2) above, imply that the conjunction $\bar{L}_i$ can be restricted to the labels ℓ such that $\langle\beta|\ell\|\alpha\rangle$. But for such labels, if $\models\langle\beta|[L]\alpha[\ell]\rangle$ then this can actually be derived, using F_1 or F_2 and the fact that $\models\langle\beta|L\rightarrow\ell\rangle$ (a consequence of property (B)). This implies that $\vdash\langle\beta|[L]\alpha[\bar{L}_1\vee\ldots\vee\bar{L}_n]\rangle$ holds and hence that $\vdash\langle\beta|[L]\alpha[\bar{L}]\rangle$

Theorem 5.2. $M\models\langle\beta|[p]\alpha[q]\rangle\Rightarrow PS(M)\vdash\langle\beta|[p]\alpha[q]\rangle$, where $p,q\in F(1)$ and α is atomic.

Proof. Lemma 5.3 and theorem 5.1 (and an application of I).

Definition. Let $p\in F(1)$, $\alpha,\beta\in PROG(tl)$, α atomic and $\langle\beta|\alpha\rangle$. Then $L^{\circ}(p,\langle\beta|\alpha\rangle)$ is the set of labels whose truth-values are not determined by p; i.e., it is the set $\{\ell\mid\models\langle\beta|[p]\alpha[\ell]\rangle,\ \not\models\langle\beta|[p]\alpha[\neg\ell]\rangle\}$.

The following lemma states the basic property of such labels:

Lemma 5.4. Let $K\in F(1)$, $p,q\in F(tl)$, $\alpha,\beta\in PROG(tl)$, α atomic and $\langle\beta|\alpha\rangle$. Assume that $M\models\langle\beta|\alpha^{\cdot}\rightarrow\neg K\rangle$, $LAB(p)=\emptyset$ and $LAB(q)\subseteq L^{\circ}(p,\langle\beta|\alpha\rangle)$.
Then $M\models\langle\beta|[K\wedge p]\alpha[q]\rangle$ implies $M\models\langle\beta|[p]\alpha[q]\rangle$.

---

[1]Note, that the pre-assertion L states nothing about the values of the program variables.

Proof. Let $L=L^{\circ}(p,\langle\beta|\alpha\rangle)$. Suppose $\not\models\langle\beta|[p]\alpha[q]\rangle$. Then, there is a transition $(\sigma,\sigma')\in Trs(\beta,\alpha)$ such that $\sigma\models p$ and $\sigma'\not\models q$. By assumption, there is a state $\bar{\sigma}$ such that $\beta\text{-}sg(\bar{\sigma})$ and $\bar{\sigma}\models\alpha^{\cdot}\wedge K$. As $LAB(p)=\emptyset$, $\bar{\sigma}\models\alpha^{\cdot}\wedge K\wedge p$ holds, too. Now consider the states $\bar{\sigma}'$ such that $(\bar{\sigma},\bar{\sigma}')\in Trs(\beta,\alpha)$. By definition of L, there is a $\bar{\sigma}'$ such that $\bar{\sigma}'|(LUVar)=\sigma'|(LUVar)$. Because $LAB(s)\subseteq L$, this means that $\bar{\sigma}'\not\models q$ and hence that $\not\models\langle\beta|[K\wedge p]\alpha[q]\rangle$; contradiction.

Finally, completeness for general transition formulae, $\langle\beta|[p]\alpha[q]\rangle$ $(p,q\in F(tl))$ can be shown. The intuition behind the proof is simple: Consider all (consistent) control locations in α. For each control location, L, derive from p the constraints on the variable values at this control point. Next, derive from q the constraints on the variable values in any state that can be reached by a single transition, when control resides at L. Finally, show that these constraints are met, if the transition is taken from a state satisfying p, too. Observe that this strategy corresponds to the way in which analogous program properties are proved in the temporal logic proof system of Manna and Pnueli [16].

Theorem 5.3. $M\models\langle\beta|[p]\alpha[q]\rangle\Rightarrow PS(M)\vdash\langle\beta|[p]\alpha[q]\rangle$, $p,q\in F(tl)$.

Proof. Because of the C_0-rules, it suffices to show completeness for atomic α. Let $L\overset{D}{=}LD(\beta)\wedge\alpha^{\cdot}\wedge\wedge\{\ell\vee\neg\ell\mid\ell\in LAB(p)\setminus LAB(\beta)\}$. By definition $\models\langle\beta|\alpha^{\cdot}\rightarrow L\rangle$. It suffices to show that $\vdash\langle\beta|[p\wedge L]\alpha[q]\rangle$, because:

$$\dfrac{[\mathrm{true}]\alpha[\alpha^{\cdot}] \qquad \alpha^{\cdot}\rightarrow(\alpha^{\cdot}\vee L)}{[\mathrm{true}]\alpha[\alpha^{\cdot}\vee L]}\,I \qquad \alpha^{\cdot}\rightarrow L \qquad [p\wedge L]\alpha[q]}{[p]\alpha[q]}\,S$$

Now, bring L into disjunctive normal form $L_1\vee\ldots\vee L_n$. Because of the C_v-rule, concentrate on $\vdash\langle\beta|[p\wedge L_i]\alpha[q]\rangle$. By definition, $LAB(p)=LAB(L_i)$. Hence, as in the proof of lemma 5.2, formulae $p_i\in F(t)$ can be (effectively) found, that satisfy $LAB(p_i)=\emptyset$ and $\models\langle\beta|L_i\rightarrow(p\leftrightarrow p_i)\rangle$. Consequently, it suffices to derive $\langle\beta|[p_i\wedge L_i]\alpha[q]\rangle$.

As the next step, define $M_i\overset{D}{=}sp(L_i,\langle\beta|\alpha\rangle)$. By theorem 5.2 (and I), $\vdash\langle\beta|[p_i\wedge L_i]\alpha[M_i]\rangle$. Bring M_i into conjunctive normal form $M_{i1}\wedge\ldots\wedge M_{im}$. Let $\bar{M}_i$ be the conjunction of those M_{ij} that are of the form ℓ or $\neg\ell$ (or true if no such M_{ij} exist). Clearly $\vdash\langle\beta|[p_i\wedge L_i]\alpha[\bar{M}_i]\rangle$. Again, construct formulae $q_i\in F(tl)$

that satisfy $\models\langle\beta|\bar{M}_i\rightarrow(q\leftrightarrow q_i)\rangle$ and $LAB(q_i)\cap LAB(\bar{M}_i)=\emptyset$. The following derivation justifies our concentrating on proving (I): $\vdash\langle\beta|[p_i\wedge L_i]\alpha[q_i]\rangle$.

$$\dfrac{\dfrac{[p_i\wedge L_i]\alpha[q_i] \qquad [p_i\wedge L_i]\alpha[\bar{M}_i]}{[p_i\wedge L_i]\alpha[q_i\wedge\bar{M}_i]}C_{\wedge} \qquad \bar{M}_i\wedge q_i\rightarrow q}{[p_i\wedge L_i]\alpha[q]}I$$

If $\models\langle\beta|\cdot\delta\rightarrow\neg L_i\rangle$ then (I) reduces to $\langle\beta|[false]\alpha[q_i]\rangle$ which is easily proved using As or T. Next, assume that $\not\models\langle\beta|\cdot\delta\rightarrow\neg L_i\rangle$. Moreover, suppose that (II) $LAB(q_i)\subseteq L^{\circ}(L_i,\langle\beta|\alpha\rangle)$. Then, lemma 5.4 applies: As $\models\langle\beta|[p_i\wedge L_i]\alpha[q_i]\rangle$, we obtain $\models\langle\beta|[p_i]\alpha[q_i]\rangle$ and hence $\models\langle\beta|sp(p_i,\langle\beta|\alpha\rangle)\rightarrow q_i\rangle$. Because $LAB(p_i)=\emptyset$, the representation of the spc in lemma 5.2, simplifies to $\alpha^{*}\wedge LC(\beta)\wedge\bar{p}_i$, so that $\models\langle\beta|\alpha^{*}\wedge\bar{p}_i\rightarrow q_i\rangle$ holds. This is the premiss of the As or T rule. So, finally we obtain $\vdash\langle\beta|[p_i]\alpha[q_i]\rangle$, hence (using I) $\vdash\langle\beta|[p_i\wedge L_i]\alpha[q_i]\rangle$.

It remains to show that (II) actually holds; i.e., that $K\stackrel{D}{=}LAB(q)\setminus LAB(\bar{M}_i)\subseteq L^{\circ}(L_i,\langle\beta|\alpha\rangle)$:
Take any $\ell\in K$. Because $\ell\notin LAB(\bar{M}_i)$, $\not\models\langle\beta|\bar{M}_i\rightarrow\ell\rangle$ and $\not\models\langle\beta|\bar{M}_i\rightarrow\neg\ell\rangle$ hold and therefore $\not\models\langle\beta|M_i\rightarrow\ell\rangle$ and $\not\models\langle\beta|M_i\rightarrow\neg\ell\rangle$. By definition of M_i, this means that $\ell\in L^{\circ}(L_i,\langle\beta|\alpha\rangle)$.

6. DISCUSSION.

The paper introduces compositionality into proofs of temporal properties of concurrent programs. The key-observation to view programs as sets of sequences of atomic actions, leads to the proposal of Transition Logic as supporting compositional reasoning. Essential features of the deductive system for this logic - as exemplified in the completeness proof - are (1) the axiomatization of the flow-of-control and (2) the expressibility of the strongest post-condition of atomic programs.

Transition Logic is developed here, for a rudimentary language, i.e., for regular programs with merge over assignments and tests. It would be interesting to consider languages with more intricate syntactic structure and introduce, e.g., recursion, synchronization commands or a construct, $\langle\alpha\rangle$, to execute arbitrary programs α, as indivisible actions. The logic can be straightforwardly extended to deal with the latter two constructs; no basic difficulties are envisaged in treating recursion.

Another possibility is introducing communication actions. Of course, there is an easy way to deal with such commands, as is illustrated by Lamport and Schneider [13] in the context of CSP. However, that solution ignores the interesting fact that for such languages (without variable sharing) there is a clear distinction between the internal behaviour of a program and its external behaviour, i.e., the sequences of communications it is willing to participate in. A distinction that should be reflected in Transition Logic, thus making a connection with [19] and [23] on partial correctness of networks of communicating processes.

Transition Logic concentrates on (input-output) properties of atomic actions. While this suffices for proving any temporal property, it is at the same time unsatisfactory because one would like to reason about temporal properties in a more direct way. The basic problem is to define the execution sequences of programs when interleaving occurs. In the current set-up, this is impossible to do in a compositional way. Hence, satisfaction of temporal formulae cannot be defined. A possible extension would be to first formulate a set of assumptions, specifying the way the environment of a program may alter the program state. Relative to such assumptions, execution sequences of programs can be defined and temporal properties can be reasoned about using temporal modalities. Such an extension of Transition Logic, would be a first step in providing Lamport's modular specification method [14] with a compositional deductive system.

Finally, an obvious question is, whether the logic can be extended to deal with just and fair executions of programs. A problem arises here: For a transition to be justly taken, it should eventually become continuously enabled. Hence, it should be certain that no move of the environment - of which nothing is known - can disable this transition from some time onwards. Similar reasoning applies to fairness, as a move need <u>not</u> be fairly taken, if that move eventually becomes continuously disabled. It follows that justness and fairness, too, only make sense relative to assumptions about the behaviour of the environment.

ACKNOWLEDGEMENTS.
I thank Willem Paul de Roever for keeping me on course while developing these ideas and for the unfailing way in which he, time and again, homed in on the weak spots in both the proofs and the presentation. This paper has benefited, too, from some remarks by Amir Pnueli.

G 50252-X

[1] K. R. ABRAHAMSON. Modal logic of concurrent nondeterministic programs. LNCS70, G. Kahn ed., pp.21-33, Springer Verlag, New York, 1979.

[2] K. R. APT, C. DELPORTE. An axiomatization of the intermittent assertion method using temporal logic (extended abstract). LNCS154, J. Diaz ed., pp.15-27, Springer Verlag, New York, 1983.

[3] H. BEKIC. Towards a mathematical theory of processes. Report TR25.125, IBM Laboratory, Vienna, 1971.

[4] E. BEST. A relational framework for concurrent programs using atomic actions. Proc. IFIP TC2 Conference, D. Bjørner ed., North-Holland, 1982.

[5] P. and R. COUSOT. On the soundness and completeness of generalized Hoare logic. Report CRIN-82-P093, Nancy, 1982.

[6] E. W. DIJKSTRA. A discipline of programming. Prentice Hall, 1979.

[7] R. W. FLOYD. Assigning meaning to programs. Proc. Symp. in Appl. Math.-19, J. T. Schwartz ed., pp.19-32, AMS, 1967.

[8] D. GRIES. The science of programming. Springer Verlag, New York, 1982.

[9] D. HAREL. First order dynamic logic. LNCS68, Springer Verlag, New York,1979.

[10] C. A. R. HOARE. An axiomatic basis for computer programming. CACM12-10, pp.576-580, 1969.

[11] T. JANSEN, P. van EMDE BOAS. Some observations on compositional semantics. LNCS131, D. Kozen ed., pp.137-170, Springer Verlag, New York, 1982.

[12] L. LAMPORT. The "Hoare logic" of concurrent programs. Acta Inf. 14, pp.21-37, 1980.

[13] L. LAMPORT, F. B. SCHNEIDER. The "Hoare logic" of CSP, and all that. Comp. Sci. Lab., SRI, 1982.

[14] L. LAMPORT. Specifying concurrent program modules. TOPLAS 5-2, pp.190-223, 1983.

[15] D. LEHMAN, A. PNUELI, J. STAVI. Impartiality, justice and fairness: the ethics of concurrent termination. LNCS115, pp.264-278, Springer Verlag, New York, 1981.

[16] Z. MANNA, A. PNUELI. How to cook a temporal proof system for your pet language. Proc. POPL, Austin, ACM, 1983.

[17] Z. MANNA, A.PNUELI. Proving precedence properties - the temporal way. LNCS154, J. Diaz ed., Springer Verlag, New York, 1983.

[18] R. MILNER. An approach to the semantics of parallel programs. Proc. of the convegno di Informatica Teorica, Pisa, 1973.

[19] J. MISRA, K. M. CHANDY. Proofs of networks of processes. IEEE SE-N, SE-7, no.4, pp.417-427, 1981.

[20] S. OWICKI, D. GRIES. An axiomatic proof technique for parallel programs I. Acta Inf. 6, pp.319-340, 1976.

[21] W. REISIG. Partial order semantics versus interleaving semantics for CSP-like languages and its impact on fairness. Proc. ICALP84, LNCS, Springer Verlag, New York, 1984.

[22] A. SALWICKI, T. MULDNER. On the algorithmic properties of concurrent programs. LNCS125, E. Engeler ed., pp.169-177, Springer Verlag, New York, 1981.

[23] J. ZWIERS, A. de BRUIN, W. P. de ROEVER. A proof system for partial correctness of dynamic networks of processes (extended abstract). Proc. 2nd Workshop on Logics of Programs, LNCS, D. Kozen, E. Clarke eds., Springer Verlag, New York, 1984.