Traces and logic

Eindhoven University of Technology

Department of Mathematics and Computing Science

Traces and Logic

by

Wojciech Penczek and Ruurd Kuiper

94/52

ISSN 0926-4515

Computing Science Report 94/52
Eindhoven, December 1994

Traces and Logic[*]

Wojciech Penczek[†]
Institute of Computer Science
Polish Academy of Sciences
Ordona 21, 01-237 Warsaw, Poland
`penczek@wars.ipipan.waw.pl`

Ruurd Kuiper[‡]
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
`wsinruur@info.win.tue.nl`

DECEMBER 1994

1 Introduction

Trace systems were introduced by Mazurkiewicz [23, 24] as semantics for concurrent systems (think, for example, about Petri Nets [26]). The reader is briefly reminded about some of the characteristics that are relevant to appreciate the logics developed in connection with them.

The most characteristic feature of trace systems is that they model the distinction between concurrency and non-deterministic choice explicitly, while at the same time abstracting from interleaving. The former is achieved through an independence relation on actions, the latter through defining traces as equivalence classes of finite sequences modulo interleaving.

One evident limitation is the restriction of the independence relation to a context-independent one. As a consequence, for example Place Transition Nets [41] can not be expressed; this can, retaining the concept of traces, be remedied by

[*]This report will appear as a chapter of The Book of Traces, edited by V. Diekert and G. Rozenberg, to be published by World Scientific Publishing in Singapore.

[†]Partially supported by De stichting informatica-onderzoek in Nederland (SION) during the author's stay at Eindhoven University of Technology.

[‡]Partially supported by ESPRIT project P6021: "Building Correct Reactive Systems (REACT)".

going to semi-commutations [28, 29]. Another limitation is the exclusion of non-determinism between actions with the same label. To overcome this restriction, instances of actions have to be distinguished explicitly. This means going to, e.g., event structures, which amounts to leaving the realm of trace systems. The choice to allow only finite traces has the advantage of simplicity but the drawback that an additional notion, run, is required to capture infinitary properties. Intuitively, this notion incorporates the view that executions should be fair with respect to concurrently enabled actions. Analogous approaches based on infinite traces exist [10, 5].

The first area in which logics for trace systems make their appearance is formal verification using program proof systems, i.e., verification by hand.

Interleaving Set Temporal Logic, ISTL, developed by Katz and Peled [16, 17] can be viewed as the starting point of logics for trace systems. Historically, their aim was to make the verification of Linear Time Temporal Logic (LTL) properties of concurrent programs easier by exploiting the idea that sequences that differ only in their interleaving are in some sense equivalent. For certain properties only one, arbitrary, representative from each equivalence class of execution sequences then needs to be considered. This clearly resembles the idea of abstraction in the concept of trace. However, at that stage the relation to traces was not made explicit.

In [34], Peled and Pnueli, as a side issue, discuss the connection between trace systems and ISTL. This paper primarily considers the other side of the trace system view, namely the ability to distinguish concurrency explicitly, and the possibilities of ISTL to reason about these matters. ISTL can thus be seen as a logic that is, maybe somewhat retrospectively, directly geared to trace systems and enables to exploit both features of that approach.

The second area in which logics and trace systems come to the fore is verification by means of model checking, i.e., fully automated verification. Programs here are essentially finite state and the logics are propositional.

The ability to abstract away from interleaving of concurrent actions was again investigated first, now with the intention to avoid the state explosion problem. Godefroid [12] presents constructions of reduced finite trace automata that generate representatives for finite executions of a program. Godefroid and Wolper [13] consider model checking of safety properties using reduced trace automata.

The further aim to likewise simplify model checking of an LTL formula against a program, possibly involving liveness properties and hence infinite sequences, is present in the approach of Peled [31]. Note that again the temporal logic itself is not a trace system logic, but that the idea of representatives is used to enhance efficiency.

As regards expressiveness, model checking for propositional ISTL could be considered. As yet, this has not been investigated.

Apart from the linear logics mentioned so far, various others exist, for instance ones that deal with branching. They will make their appearance in the rest of the chapter. A state-of-the-art overview of logics and results about them like the ones just discussed is presented in the conclusions. Further topics as well as the organization of the rest of the chapter are as follows. Didactic rather than historical considerations dictated the organization of the material.

Section 2.1 contains preliminaries: trace systems together with the associated models and acceptors. Furthermore, Elementary Petri Nets are chosen as the representation of finite state concurrent systems. Also the two running examples, taken from [34], are introduced. In Section 3 various temporal logics for trace systems are compared as to their ability to distinguish between models. Section 4 deals with model checking from the efficiency point of view: LTL without the next step operator. Section 5 adapts the efficient model checking to branching time logics without the next step operator. Section 6 again concerns model checking, but now from the expressiveness point of view: CTL_P. In Section 7 program proof systems for ISTL with past operators are presented, considering both expressiveness and efficiency. Section 8 contains an axiomatization of an essential subset of ISTL with past operators. Section 9 contains some conclusions.

2 Trace Systems

This section provides notions used in the rest of the chapter. For a comparison of trace structures with other models for concurrency see [49].

2.1 Traces

The starting point is an *independence alphabet*: an ordered pair (Σ, I), where Σ is a finite set of symbols (*actions*) and $I \subseteq \Sigma \times \Sigma$ is a symmetric and irreflexive binary relation on Σ (the *independence* relation). Actions not related are said to be *dependent*.

Let $\equiv$ be an equivalence relation defined on Σ^* by $u \equiv u'$, if there is a finite sequence of action sequences $u_1, \ldots, u_n$ such that $u_1 = u$, $u_n = u'$, and for each $i < n$, $u_i = sabt$, $u_{i+1} = sbat$, for some $(a, b) \in I$ and $s, t \in \Sigma^*$. The *traces* over (Σ, I) are the equivalence classes of $\equiv$. The trace of which a string u is a representative is denoted by $[u]$. The following notations are used.

- $[\Sigma^*] = \{[u] \mid u \in \Sigma^*\}$ - the set of all traces over (Σ, I),
- $[\Sigma] = \{[a] \mid a \in \Sigma\}$ - the set of all traces corresponding to actions.

2.2 Operations on Traces

Concatenation of traces $[u], [t]$, denoted $[u][t]$, is defined as $[ut]$.

The *successor* relation $\rightarrow$ in $[\Sigma^*]$ is defined by: $\tau_1 \rightarrow \tau_2$ iff there is $a \in \Sigma$ such that $\tau_1[a] = \tau_2$. The *prefix* relation $\leq$ in $[\Sigma^*]$ is defined as the reflexive and transitive closure of the successor relation, i.e., $\leq\, = (\rightarrow)^*$. The relation $<$ denotes the transitive closure of $\rightarrow$, i.e., $(\rightarrow)^+$. Let $\tau \in [\Sigma^*]$ and $Q \subseteq [\Sigma^*]$. The following notations are used.

- $\downarrow \tau = \{\tau' \in [\Sigma^*] \mid \tau' \leq \tau\}$, $\uparrow \tau = \{\tau' \in [\Sigma^*] \mid \tau \leq \tau'\}$, $\downarrow Q = \bigcup_{\tau \in Q} \downarrow \tau$, $\uparrow Q = \bigcup_{\tau \in Q} \uparrow \tau$.

A set of traces R is *prefix-closed*, if $R = \downarrow R$. A set R of traces has the *I-diamond* property, if for all $\tau \in R$, $a, b \in \Sigma$ with $(a, b) \in I$, $(\tau[a] \in R$ and $\tau[b] \in R)$ implies $\tau[ab] \in R$. Notice that $(a, b) \in I$ implies $[ab] = [ba]$ and therefore $\tau[ba] \in R$.

2.3 Connection to Computing

In this section the notions of trace system and interpreted trace system are defined. They may be viewed as representations, corresponding to two levels of abstraction, of the behaviour of concurrent systems. The concurrent systems themselves are mostly left implicit. Therefore, intuitive notions like executions are only identified in the representation.

Definition 2.1 *A* trace system $\mathcal{T}$ *over* (Σ, I) *is a prefix-closed subset of* $[\Sigma^*]$ *with the I-diamond property.*

Where no confusion is likely, (Σ, I) is omitted. This may be viewed as representing the behaviour of a concurrent system in terms of the executed actions.

To interpret the effect of actions, a set of variables $\mathcal{Y}$ and a domain of interpretation $\mathcal{D}$ can be added.

Definition 2.2 *An* interpreted trace system $(\mathcal{T}, \mathcal{I})$, *interpreted over* $(\mathcal{Y}, \mathcal{D})$, *is a trace system* $\mathcal{T}$ *together with an interpretation function* $\mathcal{I} : \mathcal{T} \longrightarrow (\mathcal{Y} \longrightarrow \mathcal{D})$.

This may be viewed as representing the behaviour of a concurrent system in a less abstract way, where also values of variables after executing the actions in a trace are specified. Throughout the chapter, where the interpretation is not relevant and extension straightforward the exposition just considers trace systems.

A trace represents a partial execution of a concurrent system. Executions with different ordering of dependent actions yield different traces; executions that differ only in the ordering of independent actions yield the same trace.

Omitting from the definition of I-diamond the requirement that the actions involved are from I yields the following notion. A set R of traces has the *diamond* property, if for all $\tau \in R$, $a, b \in \Sigma$ with $a \neq b$, $(\tau[a] \in R$ and $\tau[b] \in R)$ implies $(a, b) \in I$ and $\tau[ab] \in R$. Considering the fact that in trace systems closed diamonds only occur for actions from I, this notion is used to indicate that for some subset of traces diamonds only open with actions from I and that the trace closing the diamond is present in the subset. A *run* R of $\mathcal{T}$ is a maximal (with respect to the inclusion ordering) subset of $\mathcal{T}$ with the diamond property. A run represents a complete execution of $\mathcal{T}$, providing all the traces that occur in the different interleavings for that execution.

A sequence $x = \tau_0 a_0 \tau_1 a_1 \ldots$ in $S \subseteq \mathcal{T}$ such that $\tau_i[a_i] = \tau_{i+1}$ for all $i \geq 0$ is a *path* in S. A path in S is *maximal* if is either infinite or its last trace does not have a successor in S. $x/\Sigma \overset{def}{=} a_0a_1\ldots$ is the *restriction of x to actions*. $x/\mathcal{T} \overset{def}{=} \tau_0\tau_1\ldots$ is the *restriction of x to traces*. Where no confusion is likely, restrictions are referred to by just x as well. $\downarrow x = \{\tau \in \mathcal{T} \mid \tau \leq \tau_i, \text{ for } i \geq 0\}$ denotes the set of traces dominated by a path x. An ***observation*** of a run R of $\mathcal{T}$ is any maximal path x in R such that $R = \downarrow x$. Note that an observation is a path which is cofinal with some run. Thus, it carries information about all actions executed in the run. A maximal path is called an observation (of $\mathcal{T}$) if it is an observation of some run in $\mathcal{T}$. A suffix $\tau_i\tau_{i+1}\ldots$ of an observation x is said to be an observation starting at τ_i. An observation, like a run, represents a complete execution of $\mathcal{T}$, now by providing all traces that occur in just one interleaving for that execution. An arbitrary path in a run might not enable to recover the run, as not even all actions occurring in the run need to be present.

The following Lemma provides an alternative characterization for observations, independent of the notion of run.

Lemma 2.3 *A maximal path $x = \tau_0a_0\tau_1a_1\ldots$ in $\mathcal{T}$ is an observation iff*

$$(\forall a \in \Sigma)(\forall i \in I\!N)(\exists j \geq i)(\tau_j[a] \notin \mathcal{T} \textit{ or } (a, a_j) \notin I).$$

Proof: See [34] and [19]. □

Obviously, the above lemma also holds for suffixes of observations. Intuitively, if an action is independent with all actions present in a suffix of a path and could have been added at any point, it is concurrently enabled without being taken. The lemma states, that such situation does not occur in observations. The reader familiar with fair computations may notice that observations are paths that are *concurrency fair* in the sense that for each continuously concurrently enabled action either that action itself or a dependent one is eventually taken.

Note that all notions concerning trace systems are defined in terms of traces and the successor relation only. Once a trace system is given as a set of equivalence classes, i.e., subsets of action sequences, the successor relation between traces can also be obtained without recourse to (Σ, I). Namely, by defining $\tau_1 \rightarrow \tau_2$ iff there are representatives $u_1 \in \tau_1$, $u_2 \in \tau_2$ and $a \in \Sigma$ such that $u_1a = u_2$. Thus, the set of traces contains all information about a trace system, except for the independence alphabet that defines its superset. A (Σ, I) that could serve this purpose can be obtained by taking as Σ all actions that occur in the traces, and as I the pairs of actions that extend two different traces to one and the same trace. Two trace systems are considered to be identical if their sets of traces are, irrespective of (Σ, I).

Trace systems and the associated notions are illustrated by the following two examples, respectively concerning abstraction and expressiveness; an example of an interpreted trace system is given in Section 2.7.

Example 2.4 **[inevitability]** The following property [25] is one for which abstraction of interleaving is useful.

- A subset $Q \subseteq \mathcal{T}$ is *inevitable*, if each observation of $\mathcal{T}$ contains a trace in Q.

$$\begin{tikzcd}
 & {[b]} & & {[cdb]} & \\
{[\epsilon]} \arrow[r] \arrow[ur] \arrow[d] & {[c]} \arrow[r] \arrow[d] & {[cd]} \arrow[r] \arrow[ur] \arrow[d] & {[cdc]} \arrow[r] \arrow[d] & {} \\
{[a]} \arrow[r] & {[ca]} \arrow[r] & {[cda]} \arrow[r] & {[cdca]} \arrow[r] & {}
\end{tikzcd}$$

Figure 1: Trace system $\mathcal{T}_1$ with successor relation

It is not always the case that to check whether a property is inevitable, abstraction of interleaving can be applied. Only one observation of each run can be considered for stable properties (i.e., properties Q such that with each trace $\tau \in Q$, $\uparrow \tau \cap \mathcal{T} \subseteq Q$), and properties of sets of actions.

A concrete example of the latter property is as follows. Consider the independence alphabet (Σ, I).

- $\Sigma = \{a, b, c, d\}$,
- $I = \{(a,c), (c,a), (a,d), (d,a)\}$.

Figure 1 shows the trace system $\mathcal{T}_1$ over (Σ, I), together with the successor relation. $\mathcal{T}_1$ contains infinitely many finite runs R_i and one infinite run R.

- $R_i = \downarrow [(cd)^i b]$, for $i \geq 0$,
- $R = \bigcup_{i=1}^{\infty} \downarrow [(cd)^i a]$.

Every maximal path in $\mathcal{T}_1$ except for $x = [\epsilon]c[c]d[cd]c[cdc]...$ and its suffixes, which ignore a, is an observation of $\mathcal{T}_1$.

- Let $Q = \{\tau \mid a \in \Sigma(\tau) \text{ or } b \in \Sigma(\tau)\}$, where $\Sigma(\tau)$ is the set of actions occurring in τ.

  Inevitability of Q means that either a or b will be executed eventually.

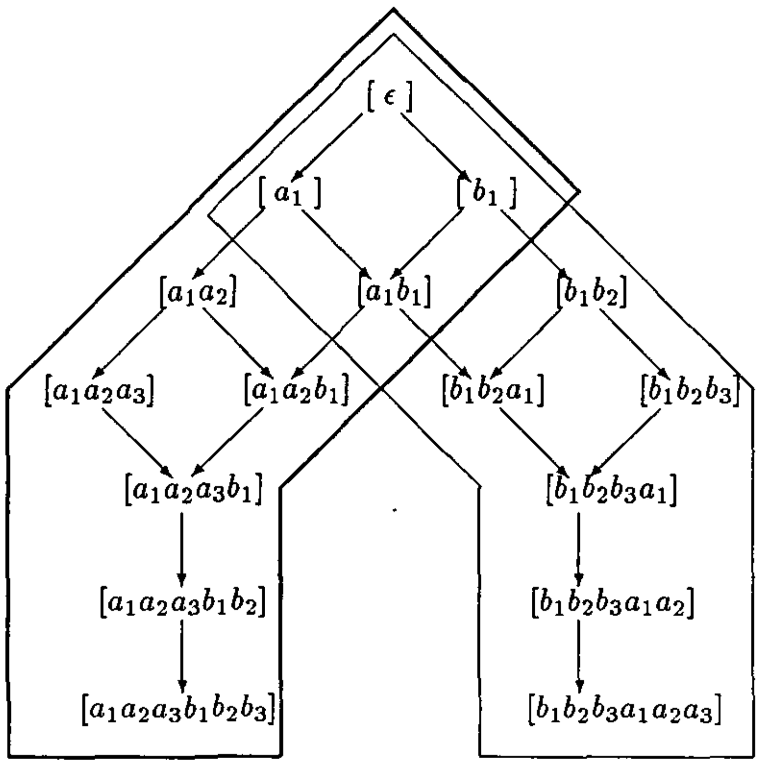


Figure 2: Trace system $\mathcal{T}_2$ with successor relation

Since the observations of the same run differ only in the ordering of independent actions, it suffices to check whether one, arbitrary, observation of each run contains a trace in Q.

It might seem surprising that Q is inevitable, as there is an infinite path $x = [\epsilon]c[c]d[cd]c[cdc]\ldots$ which does not contain any trace from Q. However, this path is not an observation, as the action a is continuously concurrently enabled and never executed, and a property is inevitable if it holds for each observation.

Example 2.5 [serializability] Serializability is a property for which expression of independence is useful. The definition of this notion in terms of trace semantics was introduced in [34].

- Two subsets $T_1 \subseteq \Sigma$ and $T_2 \subseteq \Sigma$ are *serializable* for $\mathcal{T}$, if in every run of $\mathcal{T}$, there exists an observation containing traces τ and τ' such that

  1. $T_1 \subseteq \Sigma(\tau)$, $T_2 \cap \Sigma(\tau) = \emptyset$, and $T_2 \subseteq \Sigma(\tau')$ or
  2. $T_2 \subseteq \Sigma(\tau)$, $T_1 \cap \Sigma(\tau) = \emptyset$, and $T_1 \subseteq \Sigma(\tau')$,

  i.e., all the operations of T_1 appear before those of T_2, or the other way round.

A concrete example is as follows. Consider the independence alphabet (Σ, I).

- $\Sigma = \{a_1, a_2, a_3, b_1, b_2, b_3\}$,
- $I = \{(a_1, b_1), (b_1, a_1), (a_3, b_1), (b_1, a_3), (a_1, b_3), (b_3, a_1), (a_2, b_1), (b_1, a_2), (a_1, b_2), (b_2, a_1)\}$. Note that the action a_1 commutes with all b_i actions, but that a_2 and a_3 commute with b_1 only.

Figure 2 shows the trace system $\mathcal{T}_2$ together with the successor relation. $\mathcal{T}_2$ contains two runs, marked by the thin and the thick lines, respectively.

$\mathcal{T}_2$ can be thought of as implementing a database with two transactions.

- Let $T_1 = \{a_1, a_2, a_3\}$ and $T_2 = \{b_1, b_2, b_3\}$.

Each transition represents a database read or write operation. For a correct implementation, transactions should be serializable. One can check that, indeed, in each run there is an observation satisfying the requirement of serializability, namely:

- $[\epsilon]a_1[a_1]a_2[a_1a_2]a_3[a_1a_2a_3]b_1[a_1a_2a_3b_1]b_2[a_1a_2a_3b_1b_2]b_3[a_1a_2a_3b_1b_2b_3]$ in the run marked by the thick line, and
- $[\epsilon]b_1[b_1]b_2[b_1b_2]b_3[b_1b_2b_3]a_1[b_1b_2b_3a_1]a_2[b_1b_2b_3a_1a_2]a_3[b_1b_2b_3a_1a_2a_3]$ in the run marked by the thin line.

The rest of this section mainly deals with various representations of trace systems, used in the sequel. At the end of it, a choice for a syntax for finite state concurrent systems is shown.

2.4 Frames and Models for Trace Systems

The notions of frame and model for trace systems respectively interpreted trace systems are introduced to interpret temporal logics. To start with, labelled transition systems are defined.

Definition 2.6 *A* rooted labeled transition system *(rlts) is a four-tuple $F = (W, L, \longrightarrow, w_0)$, where W is a set of states, L a set of* actions, $\longrightarrow \subseteq W \times L \times W$ *a* labelled relation *and w_0 the* root.
An interpreted rooted labeled transition system (F, V), *interpreted over $(\mathcal{Y}, \mathcal{D})$, is an rlts F together with an interpretation function $V : W \longrightarrow (\mathcal{Y} \longrightarrow \mathcal{D})$.*

Elements of $\longrightarrow$ are denoted as, e.g., $w \xrightarrow{a} w'$. When convenient, labels are omitted. Excepting L, rlts are defined up to isomorphism only.

Most of the notions given for trace systems transfer quite directly to rlts. $x = w_0a_0w_1a_1 \ldots, w_i \xrightarrow{a_i} w_{i+1}$ is a, finite or infinite, *path* in F. The *length* of x is its number of actions, denoted as $|x|$. Similar restrictions as for trace systems, to Σ and $\mathcal{T}$, now, to Σ and W, apply. A *maximal path* in an rlts is a path that is either infinite or its last state is not related to another state.

$$\begin{tikzcd}
& w_{11} & & w_{12} & \\
w_0 \arrow[ur, "b"] \arrow[r, "c"] \arrow[d, "a"] & w_{21} \arrow[r, "d"] \arrow[d, "a"] & w_{22} \arrow[ur, "b"] \arrow[r, "c"] \arrow[d, "a"] & w_{23} \arrow[r, "d"] \arrow[d, "a"] & {} \\
w_{31} \arrow[r, "c"] & w_{32} \arrow[r, "d"] & w_{33} \arrow[r, "c"] & w_{34} \arrow[r, "d"] & {}
\end{tikzcd}$$

Figure 3: The frame for trace system $\mathcal{T}_1$

Definition 2.7 *Let $F = (W, L, \rightarrow, w_0)$ be an rlts.*

- *Path $x = w_0a_0w_1 \ldots w_n$ is* accepted at w *in F, if $w_n = w$.*
- *$FinSeq(w) \overset{def}{=} \{x \mid x$ is accepted at $w\}$.*
- *$FinSeq(F) \overset{def}{=} \{FinSeq(w) \mid w \in W\ \}$.*

Frames and models are defined up to isomorphism.

Definition 2.8 *The frame F* for a trace system *$\mathcal{T}$ is an rlts for which $L = \Sigma$, $W \simeq \mathcal{T}$, $w_0 \simeq [\epsilon]$, and $w \xrightarrow{a} w'$ iff there exist τ, $\tau' \in \mathcal{T}$ such that $w \simeq \tau$, $w' \simeq \tau'$ and $\tau' = \tau[a]$.*
The model (interpreted frame) $M = (F, V)$ for an interpreted trace system *$(\mathcal{T}, \mathcal{I})$ is an interpreted rlts for which F is the frame for $\mathcal{T}$ and for all $w \in W$, $\tau \in \mathcal{T}$, $w \simeq \tau$ implies $V(w) = \mathcal{I}(\tau)$.*

When convenient, traces of $\mathcal{T}$ will be used as the names of the states of the model for $(\mathcal{T}, \mathcal{I})$.

Note that (Σ, I) is not part of the above definition. This is in line with the observation made in connection with Definition 2.1 of trace system, that no essential information is present therein.

Example 2.9 The frame for the trace system in Example 2.4 is shown in Figure 3).

The main difference between trace systems and frames is that an equivalence class of action sequences, a trace, is now presented more indirectly as the set of action sequences that is accepted at one and the same state. This can be viewed as transferring the information about independence of actions from the states, i.e., the traces, to the structure, i.e., the labelled relation.

Lemma 2.10 *Let F be the frame for trace system $\mathcal{T}$.*

- *If $w \simeq \tau$, then $\{x/\Sigma \mid x \in FinSeq(w)\} = \tau$.*
- $\{\{x/\Sigma \mid x \in FinSeq(w)\} \mid w \in W\} = \mathcal{T}$.

Proof: Directly from Definition 2.8 □

(Interpreted) trace systems and (interpreted) frames can be viewed as different ways of representing the same information. To clarify the connection it is now shown which properties need to be satisfied by rlts in order to be a frame for some trace system.

For an rlts F the relation $I_F \subseteq \Sigma \times \Sigma$ is defined as follows: $I_F = \{(a,b) \mid (\exists w, w', w'' \in W) : w' \xrightarrow{a} w,\ w'' \xrightarrow{b} w$ and $a \neq b\}$. Note that this relation is well-defined, irrespectively of whether or not the rlts corresponds to a trace system. Condition $C4$ in the Lemma below ensures that if for two actions a diamond is closed once, it is closed wherever it occurs, i.e., that I_F is context independent.

Lemma 2.11 *An rlts F is the frame for a trace system iff it satisfies the following conditions:*

C1. $W = \{w \mid w_0 \rightarrow^* w\}$ (reachability),

C2. $\{w \mid w \rightarrow w_0\} = \emptyset$ (beginning),

C3. $(\forall w, w', w'' \in W)$ *$w \xrightarrow{a} w'$ and $w \xrightarrow{a} w''$ implies $w' = w''$* (forward determinism),

C4. $(\forall w, w', w'' \in W)(\exists v \in W)$ *if $w \xrightarrow{a} w'$ and $w \xrightarrow{b} w''$ and $(a,b) \in I_F$, then $w' \xrightarrow{b} v$ and $w'' \xrightarrow{a} v$* (forward-I_F-diamond property),

C5. $(\forall w, w', w'' \in W)(\exists v \in W)$ *if $w \xrightarrow{a} w' \xrightarrow{b} w''$ and $(a,b) \in I_F$, then $w \xrightarrow{b} v \xrightarrow{a} w''$* (concurrency closure property).

C6. $(\forall w, w', w'' \in W)$ *$w' \xrightarrow{a} w$ and $w'' \xrightarrow{a} w$ implies $w' = w''$* (no auto-concurrency),

C7. $(\forall w, w', w'' \in W)(\exists v \in W)$ *if $w' \xrightarrow{a} w$ and $w'' \xrightarrow{b} w$ and $a \neq b$, then $v \xrightarrow{a} w''$ and $v \xrightarrow{b} w'$,* (backward-diamond property),

Proof: It is easy to see that the frame for a trace system satisfies the above conditions. $C1$ ensures that all states are reachable from w_0. $C2$ expresses that w_0 is the beginning. $C3$ states that each transition leaving a state has a different label. $C4$ ensures that each branching caused by independent actions can always be closed. $C5$ states that independent actions can permute. $C6$ says that each action a is dependent with itself. $C7$ expresses that if actions join, then they are independent. So, an open backward diamond can always be closed. Note that $C2$ together with $C7$ ensures that no loops occur.
It is equally straightforward, using induction on the length of action sequences where necessary, to show that each rlts satisfying the conditions $C1 - C7$ is a frame for a trace system, namely for $\{\{x/\Sigma \mid x \in FinSeq(w)\} \mid w \in W\}$ over (Σ, I_F). The full proof can be found in [38]. □

Some notions given for trace systems transfer to frames for trace systems.

Definition 2.12 *Let $F = (W, \Sigma, \rightarrow, w_0)$ be the frame for a trace system.*

a) $F' = (W', \Sigma, \rightarrow', w_0)$ is a subframe *of F if*
$W' \subseteq W$ and $\rightarrow' = \rightarrow \cap (W' \times \Sigma \times W')$.

b) A subframe F' has the forward-diamond *property, if for all $w, w', w'' \in W'$, $a, b \in \Sigma$ with $a \neq b$, $w \xrightarrow{a} w'$ and $w \xrightarrow{b} w''$ implies that there is $v \in W'$ such that $w' \xrightarrow{b} v$ and $w'' \xrightarrow{a} v$.*

c) A subframe F' is a run *of F if W' is a maximal subset of W such that F' has the forward-diamond property.*

Let $x = w_0 a_1 w_1 a_1 \ldots$ be a maximal path in a run F' of F. $\downarrow x$ denotes the set of states $\{w \in W' \mid w \rightarrow'^{*} w_i, \text{ for } i \geq 0\}$. An *observation* of a run F' of F is any maximal path x in F' such that $W' = \downarrow x$. A suffix $w_i a_i w_{i+1} \ldots$ of an observation x is said to be an observation starting at w_i.

The isomorphism $\simeq$ could be used to identify corresponding structures in the realms of trace systems respectively frames; the above notions are defined so as to give the same names to corresponding notions.

2.5 Acceptors

The rlts defined in the previous section accept sets of sequences at each state. Special rlts, frames, were shown to be equivalent to trace systems in the ability to characterize sets of sets of action sequences, i.e., sets of traces. In this section, a less restricted subclass of rlts is considered that enables, for instance, to accept trace systems that are infinite but which correspond to finite state programs, in a finite way. The idea is to enable acceptors to be more concise by allowing loops, but still requiring that the representatives of a trace are accepted at the same state. The reason for this requirement is that acceptors are to represent concurrent programs for which different interleavings of the same partial run lead to the same state. The consequences of doing so are that closure of diamonds may occur also if actions are not independent and that sequences of different length can be accepted at a state. The first consequence means in fact that the independence information can no longer be retrieved from the structure in the rlts; this is remedied by putting the independence relation back in explicitly. This enables to partition the sets of action sequences accepted at a state into traces again.

Definition 2.13 *An* acceptor *for a trace system $\mathcal{T}$ is a five-tuple $F = (W, \Sigma, \rightarrow, I, w_0)$, where $(W, \Sigma, \rightarrow, w_0)$ is an rlts and I is an independency relation in Σ, satisfying the following conditions:*

1) $\{[FinSeq(w)/\Sigma] \mid w \in W\} = \mathcal{T}$,

2) if $x \in FinSeq(w)$, then for each path y in F such that $[y/\Sigma] = [x/\Sigma]$, $y \in FinSeq(w)$, for all $w \in W$,

3) if $x \in FinSeq(w)$, then for each sequence u such that $u \in [x/\Sigma]$, there is a path y in F starting at w_0 with $y/\Sigma = u$, for all $w \in W$.

As in case of frames it is now shown which properties need to be satisfied by rlts with I in order to be an acceptor for some trace system. Notice that $C1 - C2$ and $C6 - C7$ are dropped from the characterization of a frame.

Lemma 2.14 *$F = (W, \Sigma, \rightarrow, I, w_0)$ is an acceptor of a trace system iff it satisfies the following conditions:*

C3. $(\forall w, w', w'' \in W)$ $w \xrightarrow{a} w'$ *and* $w \xrightarrow{a} w''$ *implies* $w' = w''$ *(*forward determinism*),*

C4. $(\forall w, w', w'' \in W)(\exists v \in W)$ *if* $w \xrightarrow{a} w'$ *and* $w \xrightarrow{b} w''$ *and* $(a, b) \in I$, *then* $w' \xrightarrow{b} v$ *and* $w'' \xrightarrow{a} v$ *(*forward I-diamond property*),*

C5. $(\forall w, w', w'' \in W)(\exists v \in W)$ *if* $w \xrightarrow{a} w' \xrightarrow{b} w''$ *and* $(a, b) \in I$, *then* $w \xrightarrow{b} v \xrightarrow{a} w''$ *(*concurrency closure property*).*

Proof: Follows from Definition 2.13 and the definition of trace system. □

In the literature rlts equipped with I satisfying the above conditions $C3 - C5$ are called *rooted concurrent transition systems* [43, 4] or *rooted asynchronous transition systems (rats)* [1]. The latter name will be used in this chapter.

Rats are used as acceptors for both trace systems as well as frames. As before, the connection to the former is made by defining which traces are accepted at each state of the rats. As to the latter, the frame can be viewed as being represented by the sequences accepted at each state, but now together with the relation I rather than just the structure of the frame.

Definition 2.15 *Let F be an rats.*

- $Tr(w) \overset{def}{=} \{[x/\Sigma] \mid x \in FinSeq(w)\}$.
- $Tr(F) \overset{def}{=} \{Tr(w) \mid w \in W\}$.

Note that just one trace system is accepted by an rats. This allows the following, somewhat indirect, definition of acceptor for a frame. The indirectness is caused by the fact that in a frame the relation I is encoded in the structure.

Definition 2.16 *An rats F' is an* acceptor *for a frame F if F is the frame for the trace system accepted by F'.*

To enable identifying at which state of an rats F which traces are accepted, an *acceptance function* $AC : Tr(F) \longrightarrow W$ is defined, assigning to each trace accepted by F the state of F which accepts a representative of this trace, i.e., $AC(\tau) = w$ iff for some $x \in FinSeq(w)$, $[x/\Sigma] = \tau$. Thanks to conditions $C3$ and $C5$ in Lemma 2.14, AC is well defined. AC is extended in the standard way on subsets of $Tr(F)$: $AC(P) = \{AC(\tau) \mid \tau \in P\}$, for $P \subseteq Tr(F)$.

Interpreted trace systems are accepted by rats equipped with interpretation functions.

Definition 2.17 *An interpreted rats (F, V) is an acceptor for an interpreted trace system $(\mathcal{T}, \mathcal{I})$ if $Tr(F) = \mathcal{T}$ and $V(w) = \mathcal{I}(\tau)$, for each τ such that $AC(\tau) = w$.*

Note that just one interpreted trace system is accepted by an interpreted rats. This allows the following definition of acceptor for a model.

Definition 2.18 *An interpreted rats (F, V) is an* acceptor *for a model M if M is the model for the interpreted trace system accepted by (F, V).*

2.6 Finite State Trace Systems

Since one of the aims of this chapter is to show methods of proving properties of interpreted trace systems by model checking, trace systems that are finite state are of interest. These trace systems are a subclass of the recognizable trace systems [27].

Definition 2.19 *A trace system $\mathcal{T}$ is* finite state, *if there is an equivalence relation $EQ \subseteq \mathcal{T} \times \mathcal{T}$ satisfying the following conditions.*

1 EQ has a finite index,

2 $(\forall \tau, \tau' \in \mathcal{T})(\forall \alpha \in [\Sigma])$ ((τ EQ τ' and $\tau\alpha \in \mathcal{T}$) implies ($\tau\alpha$ EQ $\tau'\alpha$)).

An interpreted trace system $(\mathcal{T}, \mathcal{I})$ is finite state, *if in addition to the above two conditions the following condition is satisfied.*

3 $(\forall \tau, \tau' \in \mathcal{T})(\tau$ EQ τ' implies $\mathcal{I}(\tau) = \mathcal{I}(\tau'))$.

The above definition states that the number of traces that are distinguishable with respect to their continuations (clause 2) and with respect to their interpretation (clause 3) is finite. However, $\mathcal{T}$ may well have infinitely many traces with different prefixes i.e., $\downarrow \tau \neq \downarrow \tau'$ for infinitely many $\tau, \tau' \in \mathcal{T}$.

Example 2.20 The trace system $\mathcal{T}_1$ of Example 2.4 is finite state. The equivalence classes of the relation $EQ \subseteq \mathcal{T}_1 \times \mathcal{T}_1$ are the following.

- $[[\epsilon]]_{EQ} = [(cd)^*]$,
- $[[a]]_{EQ} = [(cd)^* a]$,
- $[[b]]_{EQ} = [(cd)^* b]$,
- $[[c]]_{EQ} = [(cd)^* c]$,
- $[[ac]]_{EQ} = [(cd)^* ac]$.

One can build a quotient structure of $\mathcal{T}$ by EQ.

Definition 2.21 *The* quotient structure *of $\mathcal{T}$ by an equivalence relation EQ is a five-tuple $F = (W, \Sigma, \rightarrow, I, w_0)$, where:*

- *$W = \{[\tau]_{EQ} \mid \tau \in \mathcal{T}\}$ is the set of states,*
- *(Σ, I) is the given independence alphabet,*
- *$\rightarrow \subseteq W \times \Sigma \times W$ is the transition relation such that $[\tau]_{EQ} \xrightarrow{a} [\tau']_{EQ}$, if there are traces $\tau_1 \in [\tau]_{EQ}, \tau_1' \in [\tau']_{EQ}$, and $a \in \Sigma$ such that $\tau_1[a] = \tau_1'$,*
- *$w_0 = [\epsilon]_{EQ}$.*

Lemma 2.22 *The* quotient structure *of $\mathcal{T}$ by an equivalence relation EQ is an rats.*

Proof: By straightforward verification of the conditions from Lemma 2.14. □

Lemma 2.23 *A trace system has a finite acceptor iff it is finite state.*

Proof: ($\Rightarrow$) If a trace system has a finite acceptor, then a relation EQ is defined as follows: $\tau \; EQ \; \tau'$ iff $AC(\tau) = AC(\tau')$.
($\Leftarrow$) If a trace system is finite state, then from Lemma 2.22 its quotient structure with respect to EQ is a finite rats. □

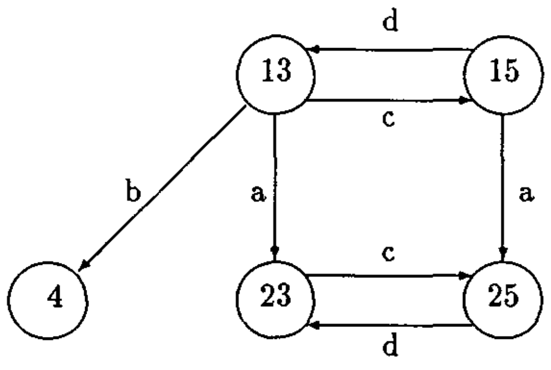


Figure 4: The rats F_1 accepting the trace system $\mathcal{T}_1$

Corollary 2.24 *A frame has a finite acceptor iff it is the frame for a finite state trace system.*

The above notions are straightforwardly extended to the interpreted case.

Example 2.25 [acceptors of trace systems] Below, the definition of a finite acceptor of the trace system $\mathcal{T}_1$ of the Example 2.4 is given. The acceptor is shown in Figure 4.

The rats $F_1 = (W_1, \Sigma_1, \rightarrow_1, I_1, w_0^1)$ accepting the trace system $\mathcal{T}_1$ is defined as follows:

- $W_1 = \{13, 15, 4, 23, 25\}$,
- $\Sigma_1 = \{a, b, c, d\}$,
- $\rightarrow_1 = \{(13, b, 4), (13, a, 23), (13, c, 15), (15, d, 13), (15, a, 25), (25, d, 23), (23, c, 25)\}$,
- $I_1 = \{(a, c), (c, a), (d, a), (a, d)\}$,
- $w_0^1 = 13$.

2.7 A Syntax for Finite State Concurrent Systems: Elementary Net-systems

Elementary Net systems [42] are a subclass of Petri Nets, namely those corresponding to finite state programming languages. They may be viewed as programs for which trace systems provide semantics; they serve as an easy representation of finite state concurrent programs. Extensions covering the infinite state case are available

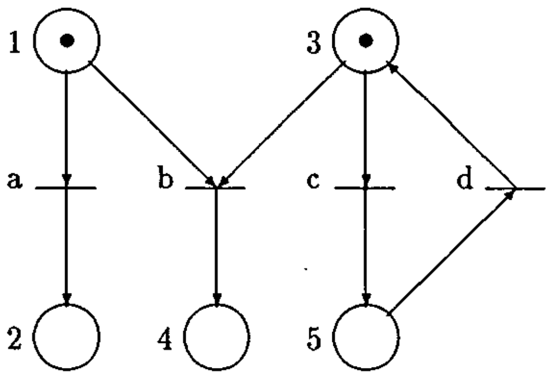


Figure 5: EN-system N_1

and extensively studied. As such extensions are not used in the present exposition, they are not considered here. For further information see, e.g., [41].

Definition 2.26 *An* Elementary Net system, *EN-system for short, is an ordered quadruple $N = (B, E, F, c_0)$, where B and E are finite, disjoint, nonempty sets of* places *and* transitions, *respectively, $F \subseteq B \times E \cup E \times B$ is the* flow *relation, with $dom(F) = B \cup E$, and c_0 is a subset of B, called the* initial case.

Any subset c of B is called a *case*. A case can be viewed as a state the EN-system is in; c_0 is then the starting state. Nets are represented graphically using lines for transitions, circles for places, and arrows for the flow relation. The initial case is represented by dots in circles (see Figure 5).

Definition 2.27 *Let N be an EN-system. For each $x \in B \cup E$, the following sets are defined:*

- $Pre(x) = \{y \mid (y, x) \in F\}$ *(*preconditions),
- $Post(x) = \{y \mid (x, y) \in F\}$ *(*postconditions),
- $Prox(x) = Pre(x) \cup Post(x)$ *(*proxconditions).

Definition 2.28 (firing sequence) *A transition t is* fire-able *at a case c and leads to the case c' (written $c[t > c')$, if $Pre(t) \subseteq c$, $Post(t) \subseteq c'$, and $c - Pre(t) = c' - Post(t)$.*

A finite sequence of transitions $u = t_0 t_1 \ldots t_n$ is a firing sequence *of N, if there is a sequence of cases $c_1, \ldots, c_{n+1}$ such that $c_i[t_i > c_{i+1}$, for $i \leq n$. This is denoted by $c_0[u > c_{i+1}$. A case c is* reachable, *if there is a firing sequence that leads to it.*

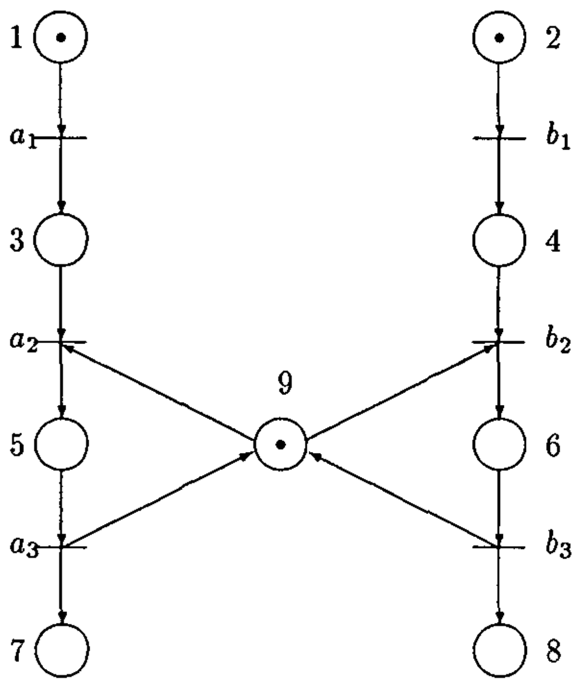


Figure 6: EN-system N_2

The last clause of the fire-ability condition prohibits firings that would fill an already occupied place, unless that place first gets emptied by the firing.

Definition 2.29 *An EN-system N is said to be* contact-free *if for each reachable case c and for all $t \in E$, the following condition holds:*

- *$Pre(t) \subseteq c$ implies $Post(t) \cap (c - Pre(t)) = \emptyset$.*

So, if a reachable case contains the preconditions of a transition, then it does not contain its postconditions, unless they coincide. Therefore, for contact-free nets a transition t is fire-able at c, just when $Pre(t) \subseteq c$.

All EN-systems used in this chapter as examples are contact-free.

Example 2.30 In Figure 5, the EN-system N_1 is represented graphically.

The EN-system $N_1 = (B_1, E_1, F_1, c_0^1)$, where

- $B_1 = \{1, 2, 3, 4, 5\}$,
- $E_1 = \{a, b, c, d\}$,
- $F_1 = \{(1, a), (a, 2), (1, b), (b, 4), (3, b), (3, c), (c, 5), (5, d), (d, 3)\}$,

3 Comparison of Temporal Logics on Trace Systems

In this section temporal logics are introduced. These are propositional versions of:

- Linear Time Temporal Logic (LTL),
- Computation Tree Logic$^{(*)}$ (CTL$^{(*)}$),
- Interleaving Set Temporal Logic$^{(*)}$ (ISTL$^{(*)}$),
- Computation Tree Logic with Past Operators (CTL$_P$).

These four logics are considered because they have been interpreted on models for interpreted trace systems. Moreover, they allow us to show the main advantages of using the trace semantics. For LTL and CTL it is shown that model checking can be more effective if trace semantics is applied. For ISTL and CTL$_P$ it is shown that new important properties can be expressed and proved either by proof rules or by model checking, respectively.

The logics differ mainly in the way they are interpreted over models for interpreted trace systems. LTL is interpreted over all paths of a model, CTL$^{(*)}$ over the tree defined by a model, ISTL over all runs of a model, and CTL$_P$ over the whole partial order structure of a model.

Next, formal definitions of the logics are given and their distinguishing power between models for trace systems is compared. It is shown which logics can distinguish branching points and which can distinguish concurrency from non-determinism.

3.1 Collection of Logics

In this section the syntax and semantics are defined for LTL, CTL*, ISTL*, and CTL$_P$. Propositional versions of these logics are defined on models for interpreted trace systems over $(\mathcal{Y}, \mathcal{D})$, where $\mathcal{Y}$ is equal to the set of propositional variables PV and $\mathcal{D}$ is equal to the set of Boolean variables $\{true, false\}$. It is convenient to start with the definition of CTL*.

3.1.1 Computation Tree Logic* (CTL*)

The language of CTL* [7, 3] is composed of state and path formulas. As the names indicate, state formulas are interpreted over states and path formulas are interpreted over paths.

Syntax of CTL*

Syntactically, the languages of all other logics considered in this paper are extensions or restrictions of this well-known language.

The set of state formulas and the set of path formulas is defined inductively:

S1. Each $q \in PV$ is a state formula,

S2. if φ and ψ are state formulas, then so are $\neg\varphi$ and $\varphi \wedge \psi$,

S3. if φ is a path formula, then $E\varphi$ is a state formula,

P1. any state formula φ is also a path formula,

P2. if φ, ψ are path formulas, then so are $\varphi \wedge \psi$ and $\neg\varphi$,

P3. if φ, ψ are path formulas, then so are $X\varphi$ and $(\varphi U \psi)$.

E is a path quantifier with the intuitive meaning: there is a path. X is the next step operator and U denotes Until.

The following abbreviations will be used for all the logics discussed:

- $\varphi \vee \psi \overset{def}{=} \neg(\neg\varphi \wedge \neg\psi)$; $true \overset{def}{=} \varphi \vee \neg\varphi$, for any φ; $\varphi \rightarrow \psi \overset{def}{=} \neg\varphi \vee \psi$,
- $\varphi \oplus \psi \overset{def}{=} (\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi)$; $\varphi \equiv \psi \overset{def}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$,
- $F\varphi \overset{def}{=} trueU\varphi$; $A(\varphi U \psi) \overset{def}{=} \neg(E(\neg\psi U(\neg\varphi \wedge \neg\psi)) \vee EG(\neg\psi))$,
- $AG\varphi \overset{def}{=} \neg EF\neg\varphi$, $AH\varphi \overset{def}{=} \neg EP\neg\varphi$.

Semantics of CTL*

Let $M = ((W, \Sigma, \rightarrow, w_0), V)$ be a model and $p = x_0 a_0 x_1 a_1 \ldots$ be a maximal path starting at $x_0 \in W$. Let p_i denote the suffix $x_i a_i x_{i+1} a_{i+1} \ldots$ of p.

S1. $x \models q$ iff $V(x)(q) = true$, for $q \in PV$,

S2. $x \models \neg\varphi$ iff not $x \models \varphi$,

$x \models \varphi \wedge \psi$ iff $x \models \varphi$ and $x \models \psi$,

S3. $x \models E\varphi$ iff $p \models \varphi$ for some path p starting at x,

P1. $p \models \varphi$ iff $x_0 \models \varphi$ for any state formula φ,

P2. $p \models \varphi \wedge \psi$ iff $p \models \varphi$ and $p \models \psi$,

$p \models \neg\varphi$ iff not $p \models \varphi$,

P3. $p \models X\varphi$ iff $p_1 \models \varphi$,

$p \models (\varphi U \psi)$ iff $(\exists i \geq 0)\, p_i \models \psi$ and $(\forall j :\ 0 \leq j < i)\, p_j \models \varphi$.

A CTL* formula φ is said to be valid in a model M (written $M \models_{CTL^*} \varphi$) iff $M, w_0 \models \varphi$.

Example 3.1 Let $M = (F, V)$ be the model for an interpreted trace system $(\mathcal{T}, \mathcal{I})$. For instance, the following properties can be expressed in CTL*:

- $M \models_{CTL^*} AG\varphi$ – φ is an invariant in $(\mathcal{T}, \mathcal{I})$,
- $M \models_{CTL^*} AF\varphi$ – φ will eventually hold in $(\mathcal{T}, \mathcal{I})$,
- $M \models_{CTL^*} EF\varphi$ – φ is possible in $(\mathcal{T}, \mathcal{I})$.

3.1.2 Computation Tree Logic (CTL)

The language of the logic CTL is the restriction of the language of CTL* such that only one single linear time operator (F,G,X, or U) can follow a path quantifier (A or E).

3.1.3 Syntax of CTL

The set of CTL formulas is the maximal one generated by the rules:

S1. Each $q \in PV$ is a formula,

S2. if α and ψ are formulas, then so are $\neg\varphi$ and $\varphi \wedge \psi$,

S3'. if φ, ψ are formulas, then so are $EX\varphi$, $EG\varphi$, and $E(\varphi U \psi)$.

The semantics is the subset of the semantics of CTL* concerning CTL formulas.

Example 3.2 The following are CTL* formulas, which are not expressible in CTL:

- $AFG\varphi$,
- $AGF\varphi$.

3.1.4 Linear Time Temporal Logic (LTL)

The language of the logic LTL [22] is the restriction of the language of CTL* such that it does not contain path quantifiers.

Syntax of LTL

The set of LTL formulas is the maximal one generated by the rules $S1 - S2$ and $P1 - P3$.

Semantics of LTL

LTL formulas are interpreted over models corresponding to executions of concurrent systems. These correspond to maximal paths through interpreted trace systems in the present framework.

The semantics is the subset of the semantics of CTL* concerning LTL formulas.

An LTL formula φ is said to be valid in a model M (written $M \models_{LTL} \varphi$) iff $p \models \varphi$, for all maximal paths p starting at w_0 in M.

Example 3.3 Let $M = (F, V)$ be the model for an interpreted trace system $(\mathcal{T}, \mathcal{I})$. For instance, the following properties can be expressed in LTL:

- $M \models_{LTL} G\varphi$ - φ is an invariant in $(\mathcal{T}, \mathcal{I})$,
- $M \models_{LTL} F\varphi$ - φ is inevitable in $(\mathcal{T}, \mathcal{I})$.

3.1.5 Interleaving Set Temporal Logic* (ISTL*)

ISTL* was introduced in Peled's thesis [30] and then extensively studied in [16, 17]. The logic enables explicit reasoning about the observations and the runs they belong to.

Syntax of ISTL*

The syntax of ISTL* is the same as that of CTL*.

Semantics of ISTL*

Although the syntax of ISTL* is the same as that of CTL*, the semantics is different. ISTL* formulas are interpreted over models for runs of interpreted trace systems.

Let $M = (F, V)$ be the model for an interpreted trace system $(\mathcal{T}, \mathcal{I})$. $M' = (F', V')$ is an ISTL* *model* for M if F' is a run of F and $V' = V|W'$.

One can easily notice that for each ISTL* model M' there is a run R of $\mathcal{T}$ such that M' is the model for the interpreted run $(R, \mathcal{I}|R)$.

Let O_{x_0} be the set of all observations of M' starting at $x_0 \in W'$ and let $o = x_0 a_0 x_1 a_1 \ldots \in O_{x_0}$. Let o_i denote the suffix $x_i a_i x_{i+1} a_{i+1} \ldots$ of o.

S1, S2, P1 - P3 like for CTL*,

S3. $x_0 \models E\varphi$ iff $o \models \varphi$ for some observation $o \in O_{x_0}$ starting at x_0.

An ISTL* formula φ is said to be valid in a model M (written $M \models_{ISTL^*} \varphi$) iff $M', w_0 \models \varphi$, for all ISTL* models M' for M.

Example 3.4 In addition to the properties expressible in LTL (but now quantified over observations) formulas of ISTL* can express partial order properties.

- $M \models_{ISTL^*} AG\varphi$ – φ is an invariant in $(\mathcal{T}, \mathcal{I})$,

- $M \models_{ISTL^*} AF\varphi$ – φ is inevitable in $(\mathcal{T},\mathcal{I})$ (under the concurrency fairness assumption),
- $M \models_{ISTL^*} EF\varphi$ – φ holds at some state of each run in $(\mathcal{T},\mathcal{I})$.

It will be show in the next section that serializability can be expressed in ISTL*.

3.1.6 Interleaving Set Temporal Logic (ISTL)

The language of ISTL is the same as that of CTL. Therefore, the logic ISTL is a restriction of ISTL* in the same manner as the logic CTL was a restriction of CTL*. Again, only a single linear time operator (F,G,X, or U) can follow a path quantifier (A or E).

3.1.7 Computation Tree Logic with Past Operators (CTL_P)

CTL_P was introduced in [39, 36] in order to reason about partial order properties. The language of CTL_P is an extension of the language of CTL by past operators. The original definition of CTL_P included next step modalities labelled with actions. It is further shown that if a valuation function encodes names of action executed between successive states, then the labelled modalities are expressible using the unlabelled versions.

Syntax of CTL_P

Now, the set of CTL_P formulas is defined.

S1. Each $p \in PV$ is a formula,

S2. if φ and ψ are formulas, then so are $\varphi \wedge \psi$, $\neg\varphi$,

S3. if φ is a formula, then so are $EX\varphi$, $EG\varphi$, and $E(\varphi U \psi)$,

S4. if φ is a formula, then so are $EY\varphi$ and $EP\varphi$.

The symbols E and A can be called observation quantifiers (they correspond to path quantifiers in CTL). The other symbols have the following intuitive meaning: X - next step, U - Until, Y - backward step, P - sometime in the past. Formulas of the form $EY\varphi$ and $EP\varphi$ are called past formulas. Notice that the past formulas are not symmetric to the future formulas. This is motivated by the desire to define the simplest logic enabling to express partial order properties. Extensions with $EH\varphi$ and $E(\varphi S \psi)$ are possible, but will not be discussed in this chapter.

Semantics of CTL$_P$

Let $M = ((W, \Sigma, \rightarrow, w_0), V)$ be a model and $x \in W$. The notion of *truth* in M is defined by the relation $\models$ as follows:

S1. $M, x \models q$ iff $V(x)(q) = true$, for $q \in PV$,

S2. if φ, ψ are state formulas,

$M, x \models \neg\varphi$ iff not $M, x \models \varphi$,

$M, x \models \varphi \wedge \psi$ iff $M, x \models \varphi$ and $M, x \models \psi$,

S3. $M, x \models EX\varphi$ iff $M, x' \models \varphi$ for some $x' \in W$ with $x \rightarrow x'$,

$M, x_0 \models EG\varphi$ iff there is an observation $o = x_0 a_0 x_1 a_1 \ldots$ such that for all $i \geq 0$ $M, x_i \models \varphi$,

$M, x_0 \models E(\varphi U \psi)$ iff there is an observation $o = x_0 a_0 x_1 a_1 \ldots$ and $k \geq 0$ such that $M, x_k \models \psi$, and for all $0 \leq i < k$: $M, x_i \models \varphi$,

S4. $M, x \models EY\varphi$ iff $M, x' \models \varphi$, for some $x' \in W$ with $x' \rightarrow x$,

$M, x \models EP\varphi$ iff $M, x' \models \varphi$, for some $x' \in W$ with $x' \rightarrow^* x$.

A formula φ is *valid in a model M* (written $M \models_{CTL_P} \varphi$), if $M, w_0 \models \varphi$. A formula φ is said to be *valid*, if $M \models \varphi$, for all models M.

The language of CTL$_P$ contains all the CTL formulas (with slightly different semantics, tuned to observations) and moreover the formulas with the past modalities EP and EY.

Example 3.5 In addition to the properties expressible in CTL (but quantified now over observations), CTL$_P$ formulas can express partial order properties, i.e., properties requiring the distinction between concurrency and non-determinism.

- $M \models_{CTL_P} AG\varphi$ – φ is an invariant in $(\mathcal{T}, \mathcal{I})$,
- $M \models_{CTL_P} AF\varphi$ – φ is inevitable in $(\mathcal{T}, \mathcal{I})$, (under a concurrency fairness assumption),
- $M \models_{CTL_P} EF\varphi$ – φ is possible in $(\mathcal{T}, \mathcal{I})$,
- $M \models_{CTL_P} EF(AH\varphi)$ – there is a partial execution in $(\mathcal{T}, \mathcal{I})$ such that φ holds at all its states,
- $M \models_{CTL_P} EG(AH\varphi)$ – there is a run in $(\mathcal{T}, \mathcal{I})$ such that φ holds at all its states,
- $M \models_{CTL_P} AF(EP\varphi)$ – φ holds at some state of each run in $(\mathcal{T}, \mathcal{I})$,

- $M \models_{CTL_P} AG(\varphi \rightarrow EP\psi)$ – always if φ holds, ψ held in the past; this formula allows for specifying snapshots of concurrent programs (φ and ψ do not contain temporal formulas).

Examples of snapshots of concurrent systems are shown in [18, 33]. It is shown below that serializability can be expressed in the language of CTL$_P$.

3.2 Encoding Labelled Next Step Operators

The label of the transition between two adjacent states of a model can be encoded by the valuation of these states. This encoding allows to derive labelled next step operators EX_a and EY_a.

Let M be the model for an interpreted trace system, $PV_\Sigma = \{p_a \mid a \in \Sigma\}$ and $PV_\Sigma \subseteq PV$.

Definition 3.6 *The valuation function V* encodes actions *if for all $p_a \in PV_\Sigma$, $V(w)(p_a) = true$ iff the number of occurrences of $\xrightarrow{a}$-transitions in a path accepted at w is odd.*

The definition is correct since the paths accepted at w in M have the same numbers of $\xrightarrow{a}$- transitions. The condition in this definition enables to find the label of the transition between two adjacent states by only looking at its valuations, i.e., the label is a iff p_a holds at exactly one of the two states.

Therefore, the following abbreviations can be defined for all the logics using EX and/or EY:

- $EX_a\varphi \overset{def}{=} (p_a \rightarrow EX(\varphi \wedge \neg p_a)) \wedge (\neg p_a \rightarrow EX(\varphi \wedge p_a))$,
- $EY_a\varphi \overset{def}{=} (p_a \rightarrow EY(\varphi \wedge \neg p_a)) \wedge (\neg p_a \rightarrow EY(\varphi \wedge p_a))$.

$EX_a\varphi$ ($EY_a\varphi$) expresses that there is the a-successor (a-predecessor, resp.) state satisfying φ.

Example 3.7 It follows from the definition of the model M for a trace system $\mathcal{T}$ that $M \models EF(EY_a(true) \wedge EY_b(true))$ for $a \neq b$ expresses that actions a and b are independent, i.e., $(a, b) \in I$.

3.3 Examples of Inevitability and Serializability

Example 3.8 The partial order properties discussed in the former examples can be formally expressed in the logics ISTL and CTL$_P$. Let $PV_i = \{p_b \mid b \in B_i\}$, where B_i is the set of places of EN-system N_i, and $M_i = (F_i, V_i)$ be the model for trace semantics $(\mathcal{T}_i, \mathcal{I}_i)$ of N_i, for $i \in \{1, 2\}$ (see Examples 2.30 and 2.32).

In the following way, the property discussed in Example 2.4 can be expressed by ISTL and CTL$_P$ formulas:

INEVITABILITY (in ISTL) $M_1 \models_{ISTL} AF(p_2 \vee p_4)$,

INEVITABILITY (in CTL$_P$) $M_1 \models_{CTL_P} AF(p_2 \vee p_4)$.

In the following way, the property discussed in Example 2.5 can be expressed by ISTL and CTL$_P$ formulas:

SERIALIZABILITY (in ISTL):

1. $M_2 \models_{ISTL} (p_1 \wedge p_2 \wedge p_9) \rightarrow EF((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9))$,
2. $M_2 \models_{ISTL} AG[((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9)) \rightarrow EF(p_7 \wedge p_8 \wedge p_9)]$.

SERIALIZABILITY (in CTL$_P$):

1. $M_2 \models_{CTL_P} (p_1 \wedge p_2 \wedge p_9) \rightarrow AFEP((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9))$,
2. $M_2 \models_{CTL_P} AG[((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9)) \rightarrow AF(EP(p_7 \wedge p_8 \wedge p_9))]$.

The first formula expresses that each run contains a state at which either control is before the execution of T_1 and after the execution of T_2, or the other way round. The second formula says that each run contains a state at which control is after the execution of T_1 and T_2.

3.4 Equivalence Notions for Frames for Trace Systems

In order to investigate equivalences imposed on models by temporal logics it is assumed that the valuation functions do not encode any structural information, but only actions executed between successive states. This is ensured by taking $PV = PV_\Sigma$ and requiring that V encodes actions (see Definition 3.6).

As before, it is possible to find the label of the transition between two adjacent states only by looking at its valuations. Let $M = (F, V)$ be a model. Notice that V satisfies the following conditions:

- $V(w_0)(p_a) = false$, for each $p_a \in PV$, and
- $V(w)(p_a) \neq V(w')(p_a)$ iff $w \xrightarrow{a} w'$.

Now, equivalence notions for frames for trace systems are formally defined.

The following definitions contain four notions of equivalences. The notion of maximal interleaving path equivalence is defined first. Let F and F' be frames for trace systems.

Definition 3.9 *F and F' are said to be* maximal interleaving path equivalent *($F \sim_{mip-e} F'$) iff $\{x/\Sigma \mid x$ is a maximal path in $F\} = \{x'/\Sigma \mid x'$ is a maximal path in $F'\}$.*

Now, Park's and Milner's notion of forward bisimulation for F and F' are given, denoted here as $F \sim_{f-b} F'$, and referred to as f-bisimulation.

Definition 3.10 *A relation $Z \subseteq W \times W'$ is an* f-bisimulation *between F and F' iff $(w_0, w'_0) \in Z$ and if $(w, w') \in Z$,*

- *if $w \xrightarrow{a} v$, then there exists $v' \in W'$ such that $w' \overset{a}{\longrightarrow}{}' v'$ and $(v, v') \in Z$, and*
- *the symmetric condition.*

F and F' are f-bisimilar *($F \sim_{f-b} F'$), if there exists an f-bisimulation between F and F'.*

Then, definitions of a run trace equivalence and backward-forward bisimulation are given.

Definition 3.11 *F and F' are* run trace equivalent *($F \sim_{r-t} F'$) iff*

- *for each run F_1 in F, there is a run F'_1 in F' such that F_1 and F'_1 are isomorphic, and*
- *the symmetric condition.*

Definition 3.12 *A relation $Z \subseteq F \times F'$ is a* bf-bisimulation *for F and F' iff Z is an f-bisimulation and if $(w, w') \in Z$,*

- *if $v \xrightarrow{a} w$, then there exists $v' \in W'$ such that $v' \overset{a}{\longrightarrow}{}' w'$ and $(v, v') \in Z$, and*
- *the symmetric condition.*

F and F' are bf-bisimilar *($F \sim_{bf-b} F'$), if there exists a bf-bisimulation for F and F'.*

3.5 Equivalences Imposed by Temporal Logics

In this section it is investigated which equivalences are imposed on models corresponding to interpreted trace systems by the different logics considered in this chapter. Each induced equivalence will be shown to coincide with one equivalence defined in the former section.

Let $M = (F, V)$ and $M' = (F', V')$ be the propositional models for interpreted trace systems $(\mathcal{T}, \mathcal{I})$ and $(\mathcal{T}', \mathcal{I}')$ over $(PV_\Sigma, \{true, false\})$, respectively. The interpretations $\mathcal{I}$ and $\mathcal{I}'$ are defined like the valuation functions in Definition 3.6.

Modal equivalences of M and M' are defined in the following way:

Definition 3.13 (modal equivalence) *The* modal equivalence *$\equiv_L$ imposed by the logic $L \in \{LTL, CTL, CTL^*, ISTL, ISTL^*, CTL_P\}$ is defined as follows:*
$M \equiv_L M'$ iff ($M \models_L \varphi \Leftrightarrow M' \models_L \varphi$) for each formula φ of the logic L.

The next four theorems match equivalences of frames for trace systems with those induced by the considered temporal logics.

The equivalence induced by LTL coincides with maximal interleaving path equivalence.

Theorem 3.14 *F and F' are maximal interleaving path equivalent iff $M \equiv_{LTL} M'$, where ($\Leftarrow$) holds, provided the sets of maximal paths in F and F' are finite in size.*

The equivalence induced by CTL$^{(*)}$ coincides with forward bisimulation.

Theorem 3.15 *F and F' are f-bisimilar iff $M \equiv_{CTL^{(*)}} M'$,*

The equivalence induced by ISTL$^{(*)}$ coincides with run trace equivalence.

Theorem 3.16 *F and F' are run trace equivalent iff $M \equiv_{ISTL^{(*)}} M'$, where ($\Leftarrow$) holds, provided F and F' have finitely many runs.*

The equivalence induced by CTL$_P$ coincides with backward-forward bisimulation.

Theorem 3.17 *F and F' are bf-bisimilar iff $M \equiv_{CTL_P} M'$.*

The proofs of the above theorems can be found in [14, 43].

3.6 Comparing Equivalences

The aim of this section is to compare equivalences.

The following theorem shows that backward-forward bisimulation and run trace equivalence coincide with isomorphism, whereas maximal interleaving path equivalence is equal to forward bisimulation for trace systems.

Theorem 3.18 *The following equalities hold:*

- $\sim_{bf-b} = \sim_{r-e} = iso$,
- $\sim_{f-b} = \sim_{mip-e} \neq iso$.

The proof can be found in [14]. The reason for both above equalities is forward determinism of frames. The relaxation of this condition, as in case of occurrence transition systems [14], makes $\sim_{bf-b}$ stronger than $\sim_{r-e}$ and $\sim_{f-b}$ stronger than $\sim_{mip-e}$.

Example 3.19 The models M and M' (see Figure 7, where the displayed propositions hold at the states) for the following two interpreted trace systems $(\mathcal{T},\mathcal{I})$ and $(\mathcal{T}',\mathcal{I}')$ (over $(PV_\Sigma, \{true, false\})$) cannot be distinguished by any LTL and CTL* formulas.

- $\mathcal{T} = \{[\epsilon],[a],[b],[ab],[ba]\}, \Sigma = \{a,b\}, I = \emptyset$, with the obvious $\mathcal{I}$,
- $\mathcal{T}' = \{[\epsilon],[a],[b],[ab]\}, \Sigma' = \{a,b\}, I' = \{(a,b),(b,a)\}$, with the obvious $\mathcal{I}'$.

But, these models can be distinguished by ISTL and CTL$_P$ formulas.

- $M \not\models_{ISTL} EXp_a \wedge EXp_b$, $M' \models_{ISTL} EXp_a \wedge EXp_b$,
- $M \not\models_{CTL_P} EF(EYp_a \wedge EYp_b)$, $M' \models_{CTL_P} EF(EYp_a \wedge EYp_b)$.

$$\begin{tikzcd}
 & \emptyset \arrow[dl, "a"'] \arrow[dr, "b"] & \\
\{p_a\} \arrow[d, "b"'] & & \{p_b\} \arrow[d, "a"] \\
\{p_a, p_b\} & & \{p_a, p_b\}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \emptyset \arrow[dl, "a"'] \arrow[dr, "b"] & \\
\{p_a\} \arrow[dr, "b"'] & & \{p_b\} \arrow[dl, "a"] \\
 & \{p_a, p_b\} &
\end{tikzcd}$$

Figure 7: Models M and M'

3.7 Notes on Expressiveness

LTL and CTL are not comparable with respect to expressiveness. For example, the LTL formula FGp is not expressible in CTL whereas the CTL formula EFp is not expressible in LTL. The situation is similar as far ISTL* and CTL$_P$ are concerned. The ISTL* formula $AFGp$ is not expressible in CTL$_P$, whereas the CTL$_P$ formula EFp is not expressible in ISTL*.

Modal logics interpreted on asynchronous transition systems have been also investigated in [21].

4 Efficient Model Checking for a Subset of LTL

Model checking for linear time temporal logic is linear in the size of the model acceptor and $PSPACE-$ *complete* in the length of the tested formula. In the context of automatic verification, a finite state concurrent system could, in principle, be given as an interpreted trace system, but, in practice, is usually given in a program-like form; for this chapter one could think of EN-systems.

Starting from a suitable description of a concurrent system, a finite interpreted rats can be obtained algorithmically that accepts all executions. However, the number of states is frequently exponential in the number of actions. Thus, the rats is often so large as to transgress the boundaries of available computing power. Therefore, methods for reducing rats are of interest as they can improve the feasibility of model checking. Unfortunately, since checking whether an LTL formula holds for an EN-system (i.e., it is true in its model) was proved to be $NP-$ *hard* in the number of the transitions of the EN-system, it is unlikely that a polynomial algorithm can be found.

In this section it is shown that if trace semantics is used, a different, smaller, structure than the rats, called a *trace automaton* suffices; various algorithms that check a formula against an rats can be applied on the trace automaton as well. The idea is that a much smaller transition graph suffices to generate for each equivalence class of sequences a representative with respect to the independence relation. Of course, this puts a constraint on the formulas to be checked: Formulas should evaluate to the same result on each representative of a class. This is called *equivalence robustness.* A, rather mild, constraint is needed to ensure equivalence robustness: Formulas should not contain next step operators. The subset of LTL formulas satisfying this property is called LTL-x.

To start with, a standard method of model checking is given, after which it is shown how the model acceptors it employs can be reduced using trace automata.

4.1 Standard Approach

Let P be an EN-system and $(\mathcal{T},\mathcal{I})$ be its trace semantics. Then, let $M_P = (F, V_P)$ be a model acceptor for the model for $(\mathcal{T},\mathcal{I})$.

The standard method of model checking of LTL [50, 47] using automata-theoretic constructions is then as follows. Since this method ignores the independence relation I, the substructure of F equal to $(W, \Sigma, \rightarrow, w_0)$ and denoted by F_P is used. To start with, F_P is turned into a generalized Büchi automaton (gba, for short) $A_P = (W, \Sigma, \rightarrow, w_0, \mathcal{F})$, where $\mathcal{F} = \{F_1, ..., F_k\} \subseteq 2^W$, by adding a set of accepting conditions. Accepting conditions are needed to accommodate fairness; they enable defining certain subsets of maximal paths (computations), called sets of *accepting computations* of F_P. A computation of A_P is accepting if for each F_j, there is some state in F_j that repeats infinitely often in the computation. $L(A_P)$ denotes the set of all accepting computations of A_P. For instance, $L(A_P)$ is equal to the set of all maximal paths of A_P, if $\mathcal{F} = \emptyset$.

Now, let $M_P = (A_P, V_P)$ be a gba with valuation function V_P and let $L(M_P)$ denote the accepting computations of M_P.

Example 4.1 The new notions are explained on the EN-system N_1 of Example 2.30. The model acceptor is defined as follows. Let $PV_1 = \{p_1, p_2, p_3, p_4, p_5\}$. $M_{N_1} = (F_1, V_1)$, where the rats F_1 is shown in Figure 4 (Example 2.25) and $V_1(w)(p_i) = \mathit{true}$ iff w corresponds to a case with the place i marked.

Assume that formulas are to be checked over observations of F_1. Since the method deals with infinite paths only, all observations need to be extended to be infinite. This can be easily achieved by adding one more transition looping at the state 4 and labelling the transition with an action name, say, b. This extension can obviously change some properties satisfied by N_1. Since the property under consideration remains unchanged, it is not discussed here how properties of extended rats can be related to properties of original ones.

Next, $\mathcal{F}_1 = \{\{4, 23\}\}$ is defined. Now, the set of accepting computations is equal to the set of observations.

Now, let φ be an LTL formula. To verify that each accepting computation of $L(M_P)$ satisfies the formula φ (written $M_P \models \varphi$), a gba with a valuation function $M_{\neg\varphi}$ should be built with $L(M_{\neg\varphi})$ equal exactly to all accepting computations satisfying $\neg\varphi$. Then $L(M_P) \cap L(M_{\neg\varphi}) = \emptyset$ should be verified.

Since M_P has got only one beginning state w_0, the above intersection may be non-empty only if $M_{\neg\varphi}$ contains a computation starting at a state with the same valuation as w_0. Therefore, first a gba with a valuation function accepting all computations satisfying $\neg\varphi$ is found and then it is restricted to its reachable part starting at the state with the same valuation as w_0. The construction used in the proof of Theorem 4.2 guarantees that there are no two states with the same valuation function.

Theorem 4.2 ([47]) *One can build a generalized Büchi automaton with a valuation function $M_\varphi = (F_\varphi, V_\varphi)$, where $F_\varphi = (W, \Sigma, \rightarrow, W_0, \mathcal{F})$, $|W| \leq 2^{O(|\varphi|)}$ such that $L(M_\varphi)$ is exactly the set of computations satisfying an LTL formula φ.*

The above theorem is a slightly modified version of the original theorem as states rather than transitions are labelled with sets of propositions. The transitions are labelled with actions of Σ.

Example 4.3 Let $\varphi_1 = F(p_2 \vee p_4)$ be a given formula to be checked over M_{N_1} of Example 4.1. Then, $\neg\varphi_1 = G(\neg p_2 \wedge \neg p_4)$. A gba with a valuation function for $\neg\varphi_1$ is then $M_{\neg\varphi_1} = ((\{w_0\}, \Sigma_1, \{(w_0, a, w_0) \mid a \in \Sigma_1\}, \{w_0\}, \{\{w_0\}\}), V_{\neg\varphi_1})$, where $V_{\neg\varphi_1}(w_0)(p_i) = false$, for $i = 2, 4$.

Next, it is checked whether $L(M_P) \cap L(M_{\neg\varphi}) = \emptyset$. This is done by taking the product of M_P and the appropriate reachable part of $M_{\neg\varphi}$ (see the definition below), and establishing whether it contains any accepting computation. If so, then M_P satisfies φ. The time complexity of the algorithm is $O(size(M_P) \times 2^{O(|\varphi|)})$.

Definition 4.4 (product) *The* product *of two generalized Büchi automata $A_1 = (W_1, \Sigma, \rightarrow_1, w_1, \mathcal{F}_1)$ and $A_2 = (W_2, \Sigma, \rightarrow_2, w_2, \mathcal{F}_2)$ is the automaton $A = (W, \Sigma, \rightarrow, w_0, \mathcal{F})$ defined by*

- $W = W_1 \times W_2$, $w_0 = (w_1, w_2)$,
- $\mathcal{F} = \bigcup_{F_j \in \mathcal{F}_1}\{F_j \times W_2\} \cup \bigcup_{F_j \in \mathcal{F}_2}\{W_1 \times F_j\}$,
- $((v_1, v_2), a, (u_1, u_2)) \in \rightarrow$ *when* $(v_1, a, u_1) \in \rightarrow_1$ *and* $(v_2, a, u_2) \in \rightarrow_2$.

A further simplification is obtained by considering only the states for which both M_P and $M_{\neg\varphi}$ have consistent valuations. This leads to the following definition for $M_{P,\neg\varphi} = (F_{P,\neg\varphi}, V_{P,\neg\varphi})$.

$F_{P,\neg\varphi}$ is equal to the product of $F_P \times F_{\neg\varphi}$ restricted to the states W', for which both components have consistent valuations, i.e., $W' = \{(v, v') \in W \times W_{\neg\varphi} \mid V_P(v) \cap Subformulas(\varphi) = V_{\neg\varphi}(v')\}$, $V_{P,\neg\varphi}((v, v')) = V_P(v)$.

Example 4.5 The product of the structures M_{N_1} and $M_{\neg\varphi_1}$ is the following structure $M_{N_1,\neg\varphi_1} = (F_{N_1,\neg\varphi_1}, V_{N_1,\neg\varphi_1})$, where $F_{N_1,\neg\varphi_1} = (\{(13, w_0), (15, w_0)\}, \Sigma_1, \{((13, w_0), c, (15, w_0)), ((15, w_0), d, (13, w_0))\}, \{(13, w_0)\}, \{\emptyset, \{(13, w_0), (15, w_0)\}\})$, (see Figure 8), and $V_{N_1,\neg\varphi_1}((xy, w_0))(p_i) = true$, for $i = x, y$.

Note that $L(M_{N_1,\neg\varphi_1}) = \emptyset$. Therefore, $M_{N_1} \models \varphi_1$.

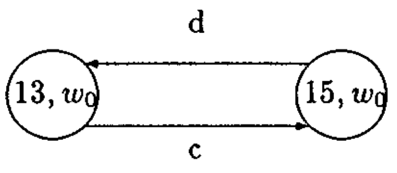


Figure 8: The product automaton $F_{N_1,\neg\varphi_1}$

4.2 Trace Approach

In order to use the power of traces in model checking the notion of trace automaton is introduced. Following that, several ways of obtaining such automata are discussed.

4.2.1 Trace Automata

Trace systems are also accepted by certain rooted labelled transition systems, called *trace automata*, which do not need to be rats and can often be smaller than these. Trace automata accept the same traces as some rats, but using a different definition of acceptance.

Definition 4.6 *A* trace automaton *is a five-tuple $TA = (W, \Sigma, \longrightarrow, I, w_0)$ such that $(W, \Sigma, \longrightarrow, w_0)$ is a rooted labelled transition system and I is an independence relation. TA* accepts *a trace τ, if there is a finite path x in TA such that $\tau \in \downarrow [x/\Sigma]$. The set of traces accepted by TA is denoted by $Tr(TA)$.*

Notice, that a trace τ is accepted by TA if TA accepts a path corresponding to a representative of a trace extending τ. This means that a trace automaton does not need to contain paths corresponding to all representatives of all accepted traces.

Lemma 4.7 *Let $F = (W, \Sigma, \longrightarrow, I, w_0)$ be an rats and $TA = (W', \Sigma, \longrightarrow', I, w_0)$ be a trace automaton such that*

- *$W' \subseteq W$, $\longrightarrow' \subseteq \longrightarrow$, and*
- *for each trace $\tau \in Tr(F)$, there is a finite path x in TA such that $\tau \in \downarrow [x/\Sigma]$.*

Then, $Tr(F) = Tr(TA)$.

Trace automata are used for model checking of certain safety properties, where they can just replace rats. The method of model checking, to be explained shortly, uses the fact that a trace automaton contains a path corresponding to a representative of each trace or its extension. For example, for checking termination it is sufficient to analyse only one representative of each maximal trace.

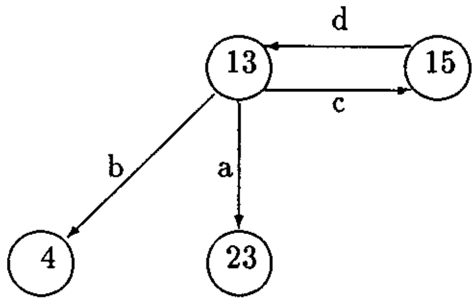


Figure 9: Trace automaton accepting the trace system $\mathcal{T}_1$

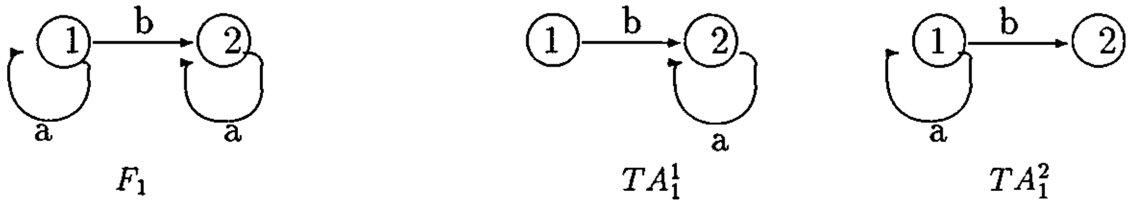


Figure 10: Rcts F_1 and two trace automata TA_1^1 and TA_1^2

Example 4.8 Figure 9 shows the trace automaton accepting the trace system $\mathcal{T}_1$ (see Example 2.4) of EN-system N_1 of Figure 5.

However, a trace automaton does not necessarily contain an infinite path corresponding to some observation of each run of the accepted trace system. Even worse, a trace automaton may not contain any infinite path corresponding to an observation (see Example 4.9). This disallows to model check liveness properties, i.e., these requiring analysis of observations (as inevitability, for example) as the standard method of model checking relies on testing whether the product of gba's with valuation functions contains an accepting computation, i.e., an infinite path satisfying some conditions. This method does not use any information given by I. So, the lack of an observation in a trace automaton might make the standard method incorrect when the rats is replaced by the trace automaton.

Example 4.9 Consider rats F_1 as given in Figure 10, with $I = \{(a,b),(b,a)\}$. Definition 4.6 allows two non-trivial reductions: TA_1^1 and TA_1^2. TA_1^1 correctly represents the infinite observation ba^ω, whereas TA_1^2 represents only the infinite path a^ω, which is not an observation

In order to also allow one checking liveness properties, trace automata have to correctly represent runs of trace systems. The way they need to represent them is similar to the way they represent traces. This means that for each run of the accepted trace system, a trace automaton should contain a path corresponding to one of its observations.

Definition 4.10 *A* trace automaton on runs *is a trace automaton TA for which $Tr(TA)$ is a trace system and which satisfies the following auxiliary property:*

- *for each run R of $Tr(TA)$, there is a path $x = w_0a_0w_1a_1\ldots$ in TA such that $[\epsilon]a_0[a_0]a_1[a_0a_1]\ldots$ is an observation of R.*

It follows from the definition that for each observation in rats F there is at least one path in TA that is an observation of the same run, if $Tr(F) = Tr(TA)$.

4.3 Algorithmically Generating Trace Automata on Runs

The main task is to obtain, in an algorithmic manner, a trace automaton on runs from some suitable description of P, say from an EN-system, without first building the whole rats. So, let N be an EN-system and $(\mathcal{T},\mathcal{I})$ be its trace semantics. Obviously, first having to construct the rats in order to obtain the trace automaton would not make the new model checking approach more efficient. Below it is described how trace theoretical ideas are used to obtain the trace automaton. The main idea is to construct the graph of a trace automaton from the EN-system by recursively expanding cases, starting from the initial case. During this construction, extra information is preserved at each case that enables to expand the graph by only some rather that all transitions that are, according to the EN-system, enabled at a case. During the construction, the graph therefore has as nodes cases plus some extra information - what kind of information and how it is represented is explained below.

There are essentially two approaches. One approach acquires the information used to determine these subsets dynamically, i.e., choices made when building the reduced automaton influence what needs to be added in subsequent construction steps. This is the *sleep set* method, developed by Godefroid (see [12]). The other approach first determines all information statically; this approach was pursued by Peled as the *ample set* method. Peled later combined the two methods (see [31]).

To simplify the exposition, in the sequel, unless stated otherwise, only concurrently fair executions are considered. This ties in with considering observations.

Definition 4.11 (concurrency fairness assumption) *For every infinite execution, for every action a that is enabled from some state on, either that action or an action dependent with it is taken in the execution at some later state.*

4.3.1 The sleep set method

In this approach, historically, rather than supplying sets of enabled actions that need to be added, sets that need not be added are used. Obviously, this is no essential difference, as complementation with respect to enabled actions immediately shows.

The idea is, that at some stage in the construction of the automaton an action can be ignored at a state if the extensions of paths through that state by that action can also be obtained as extensions of another, equivalent, path through the automaton. Note that at that stage of the expansion such extensions need not yet be available - it is enough to know that they will be available at some stage. As this information depends on what has been put into the graph in previous expansion steps, it can only be obtained during the construction of the graph, i.e., dynamically. For each node, it is represented as a set of enabled actions that need not be considered for addition; this set is called a *sleep set*.

Let N be an EN-system and $(\mathcal{T},\mathcal{I})$ be its trace semantics.

Definition 4.12 (Sleep set) *At some stage in the construction of a trace automaton* $(W,\Sigma,\rightarrow,I,w_0)$ *to accept the trace system* $\mathcal{T}$*, a set* $Sleep(w) \subseteq en(w)$ *is a* sleep set *for a case* $w \in W$ *if for each firing sequence* u *of* N *such that* $w_0[u> w$*, with* $w \xrightarrow{a} w'$ *and* $a \in Sleep(w)$*, there is a case* $v \in W$ *and a finite path* y *accepted at* v *such that* $[u][a]$ *is a prefix of* $[y/\Sigma]$.

An abstract algorithm for the construction of the graph that generates and uses the sleep sets is presented in Figure 11. The algorithm is abstract in the sense that it is highly nondeterministic: neither the order in which nodes are expanded nor the order in which actions are selected for expansion is fixed. Several different choices for this order have been investigated and various heuristics exist that improve the efficiency of the algorithm or the extend of the reduction. We shall not enter into this as the present aim is just to show how the trace theoretical ideas are used, not to present particular algorithms.

Example 4.13 Consider the trace system over $\Sigma = \{a,b\}$ and $I = \{(a,b),(b,a)\}$ accepted by the rats with two states 1 and 2, where a is continuously enabled, looping in states 1 and 2 and b is enabled only at state 1 until it is executed once, making the transition to state 2 (see Figure 10).

Clause 5 in Algorithm in Figure 11 disallows the erroneous trace automaton where the reduction consists in the removal of the loop on state 2. The sequence containing b and infinitely many $a's$ would then not be accepted.

4.3.2 The ample set method

The idea now is that enough actions need to be added at each state to ensure that every path reaching that node extends into every run. This information does not depend on previous expansion steps and is therefore obtained through a static analysis of the system. At each node, it is presented as a set of actions, indicating that only these need to be considered for addition to the graph.

Let N be an EN-system and $(\mathcal{T},\mathcal{I})$ be its trace semantics.

Definition 4.14 (Ample set) *A set* $Ample(w) \subseteq en(w)$ *is* ample *for a case* $w \in W$ *in a trace automaton* $(W,\Sigma,\rightarrow,I,w_0)$ *accepting* $\mathcal{T}$ *if for each firing sequence* u *of* N *such that* $w_0[u> w$*, for each run* R *of* $\mathcal{T}$ *such that* $[u] \in R$*,* $\{a \in \Sigma \mid [u][a] \in R\} \cap Ample(w) \neq \emptyset$.

1 Build the starting node $(w_0, \emptyset)$, where $w_0 = c_0$ - (the initial case).

Nodes $(w, Sleep(w))$ are expanded as follows. One by one, in some order that is left implicit here, all enabled transitions that are not in the $Sleep(w)$ are added.

Firstly, the case is considered where a new transition, say $w \xrightarrow{a} v$, $a \notin Sleep(w)$, is added for which the target node v was not yet present.

2 If transition $w \xrightarrow{b} v'$ with $(a, b) \in I$ is already present, then b is added to $Sleep(v)$.

Motivation: A representative for each $\sigma[ab]\sigma'$ will be generated as $\sigma[ab] = \sigma[ba]$ (as $(a, b) \in I$).

3 If $c \in Sleep(w)$ with $(a, c) \in I$, then c is added to $Sleep(v)$.

Motivation: A representative for each $\sigma[ac]\sigma''$ will be generated as $\sigma[ac] = \sigma[ca]$ (as $(a, c) \in I$) and $c \in Sleep(w)$.

Secondly, the case is considered where the expansion reuses a node that was already present.

4 If the transition does not close a loop in the expansion, i.e. it does not connect to a state which expansion has been started but not yet finished, a new sleep set is constructed as before, then the intersection of the old one and the new sleep set is taken and then the node is expanded again with this set as its sleep set.

Motivation: Reaching a node via different routes may allow to ignore different sets of transitions; only a transitions that can be ignored in both cases can safely be part of the sleep set. As this set determines which transitions need to be added, a new expansion is necessary.

5 If the transition closes a loop, then it is not added to the sleep set.

Motivation: A loop generates an infinite sequence of actions. If an action independent with all the actions in the sequence were added to the sleep set, it might be the case that some infinite sequence including infinite number of occurrences of that independent action would not be accepted by the automaton and therefore no observations of some run would be represented.

Note that 5 can still not be omitted if only concurrency fair executions are considered: The above mentioned sequence is concurrency fair.

Figure 11: Algorithm building a trace automaton on runs using sleep sets

1 Build the starting node $(w_0, Ample(w_0))$, where $w_0 = c_0$ - (the initial case),

2 If no transition from $Ample(w)$ does close a loop in the expansion, i.e. no transition does connect to a state which expansion has been started but not yet finished, then

nodes $(w, Ample(w))$ are expanded as follows. One by one, in some order left implicit here, all enabled transitions that are in the $Ample(w)$ are added. If the corresponding target node is already present in the graph it is used, otherwise it is added.

3 If a transition from $Ample(w)$ closes a loop, then all transitions enabled at w are added. If the corresponding target node is already present in the graph it is used, otherwise it is added.

Figure 12: Algorithm building a trace automaton on runs using ample sets

Thus, for each run which contains the trace $[u]$, there is an action from the ample set extending that trace within the run.

The first task is to obtain ample sets. Various approaches exist. In [31] an approach is presented where as ample sets faithful ones are taken.

Definition 4.15 *A set $F(w) \subseteq en(w)$ is* faithful *at a case w if at w until an action from $F(w)$ is executed, only actions outside $F(w)$ that are independent of all actions in $F(w)$ are enabled.*

The second task is to generate the reduced graph using a given ample set assignment. An abstract algorithm for the construction of the graph that uses the ample sets is presented in Figure 12. Again, the aim is just to show how trace theoretical ideas are used - again various heuristics exist that yield improvements.

Trace automata generated by the algorithms are not necessarily minimal, but time complexity of the algorithms is $O(|\Sigma|^2 \times n)$, where n is the number of transitions investigated when generating the trace automaton. Since it was proved that obtaining minimal trace automata is $NP - hard$, one should not aim at that.

Remarks

The different ways in which sleep sets respectively ample sets are used, the former expressing which actions need not, and the latter expressing which actions need to be added are just historically determined. Interpreting the sleep set as expressing that from the enabled actions, say en(w), those not in the sleep set need to be added makes comparison more straightforward: $en(w) \setminus Sleep(w)$ can than be compared to $Ample(w)$ and $en(w) \setminus Ample(w)$ to $Sleep(w)$. The two approaches differ in an essential way: The above correspondences do not always yield ample from sleep sets

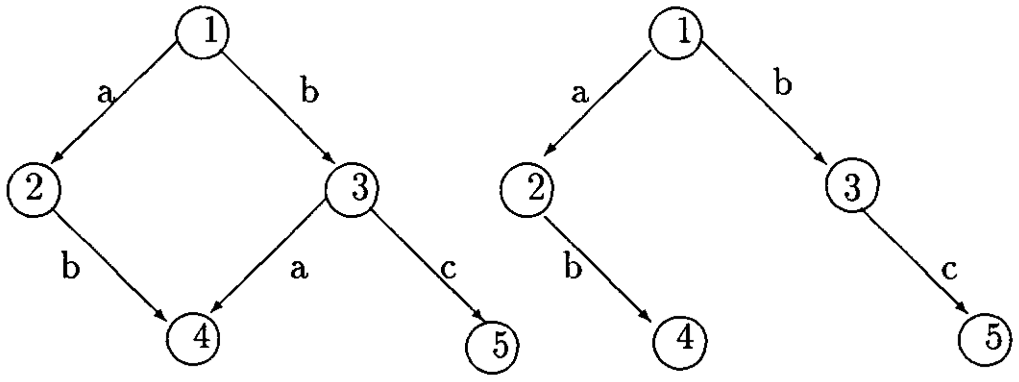


Figure 13: Sleep set reduction that is not an ample set reduction

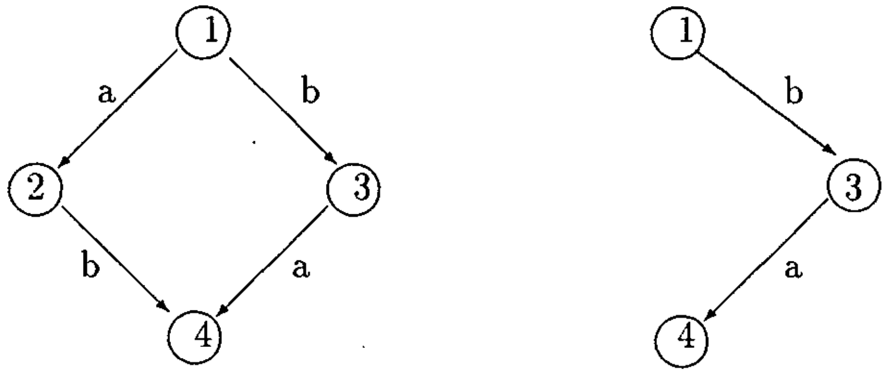


Figure 14: Ample set reduction that is not a sleep set reduction

or vice versa. The difference is even stronger: not every reduction obtained by the sleep set approach can be obtained by the ample set approach or vice versa.

Figure 13 shows an example of this fact starting with a sleep set reduction. For sleep set $\mathit{Sleep}(3) = \{a\}$, the corresponding set is $\mathit{en}(3) \setminus \mathit{Sleep}(3) = \{c\}$. As from state 3 extensions in both runs are possible, only $\mathit{Ample}(3) = \{a, c\}$ is allowed. This also implies, that no ample set reduction can remove a at 3.

Figure 14 takes ample sets as a starting point. For ample set $\mathit{Ample}(1) = \{b\}$, the corresponding set is $\mathit{en}(1) \setminus \mathit{Ample}(1) = \{a\}$. As at that state only $\mathit{Sleep}(1) = \emptyset$ is allowed. This also implies, that no sleep set reduction can remove a at 1.

4.3.3 Examples

Next, two examples of trace automata on runs that are generated by the ample set algorithm are given.

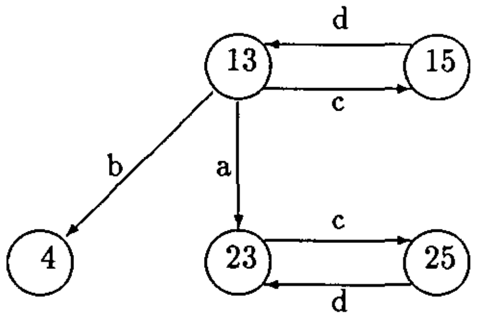


Figure 15: Trace automaton on runs accepting the trace system $\mathcal{T}_1$

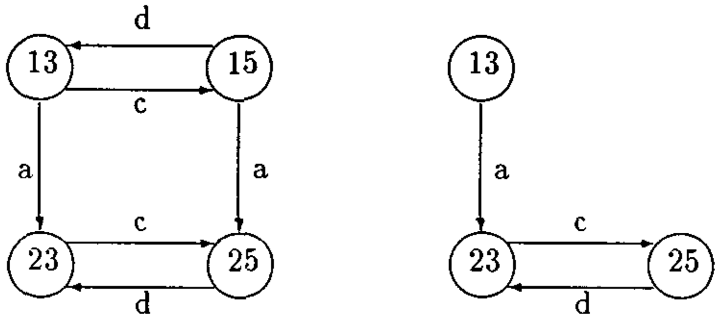


Figure 16: Rcts and trace automaton on runs accepting trace system $\mathcal{T}_1'$

Example 4.16 Figure 15 shows the trace automaton on runs accepting the trace system $\mathcal{T}_1$ (see Example 2.4) of EN-system N_1 of Figure 5.

Example 4.17 Consider EN-system N_1 from which the transition b is removed. Figure 16 shows an rats and trace automaton on runs accepting the trace semantics (without interpretation) $\mathcal{T}_1'$ of N_1. Remember that this trace automaton is built under the assumption that only concurrency fair executions need to be represented.

As mentioned in the introduction, only equivalence robust formulas are amenable to this approach. This is due to the fact that some formulas of LTL-x can hold in some, but not in all the equivalent maximal paths of models. Therefore, a model checking algorithm using a reduced structure would give another answer that the algorithm applied to the full structure.

Given a formula φ, it is possible to choose the independence relation such as to ensure equivalence robustness. Namely, by restricting the independence relation I in F_P whenever it is necessary to make equivalence classes of maximal sequences small enough for φ to be equivalence robust.

An action a is said to be *visible for a proposition* q in a structure M_P if there are two states $w, w' \in W$ such that $w \xrightarrow{a} w'$ and $V_P(w)(q) \neq V_P(w')(q)$. It is assumed that for each proposition q appearing in φ it is possible to calculate efficiently the set $Vis(q) \subseteq \Sigma$ of actions that includes at least the visible actions for q. $Vis(\varphi)$ is defined as the union of all $Vis(q)$ with q occuring in φ.

Firstly, a sufficient condition for φ to be equivalent robust is given.

Lemma 4.18 *If $(Vis(\varphi) \times Vis(\varphi)) \cap I = \emptyset$, then φ is equivalent robust.*

> The proof relies on showing that each two equivalent maximal paths with valuation function restricted to propositions appearing in φ are equivalent up to stuttering, i.e, repeating the same state finitely many times. This implies that they cannot be distinguished by any LTL-x formula.

Next, I' can be defined as $I \setminus I_\varphi$, where $I_\varphi = Vis(\varphi) \times Vis(\varphi)$, which ensures that φ can be checked over a gba with valuation function built over a trace automaton accepting $Tr(F'_P)$ with $F'_P = (W, \Sigma, \rightarrow, I', w_0)$.

However, it turns out that I_φ can be even smaller than the one defined above. This follows from the fact that equivalence robustness is preserved by Boolean operators. (Notice that it is not preserved by temporal operators.)

Therefore, if $\varphi = \varphi_1 \vee \varphi_2$ (or $\varphi_1 \wedge \varphi_2$), then I_φ can be defined as $(\Sigma(\varphi_1) \times \Sigma(\varphi_1)) \cup (\Sigma(\varphi_2) \times \Sigma(\varphi_2))$, which is usually smaller than $Vis(\varphi) \times Vis(\varphi)$.

A strategy to define I_φ as small as possible is to rewrite the formula φ in an equivalent form in which as many as possible boolean operators are not in the scope of a temporal modality. This procedure can also be automated efficiently. It is worth mentioning here that one should avoid striving for optimal rewriting as the complexity of achieving such is $NP-hard$ in the size of the formula.

Example 4.19 Notice that $\varphi_1 = F(p_2 \vee p_4)$ is equivalence robust for $M_{N_1} = (F_1, V_1)$ of Example 4.1. Therefore, a trace automaton can be used instead of F_1 in the standard way for model checking. Such a trace automaton is shown in Example 4.16.

The method presented above has been shown to be much better than the standard method when there is no very tight coupling between the concurrent components of the system. As examples indicate model checking of LTL-x using trace automata can be efficiently used in practical applications.

Remarks

An independent approach that achieves reductions has been developed by Valmari [45, 46, 51]. It is comparable to the ample set approach in that reductions are achieved by assigning *stubborn* sets to states, again based on static analysis that fulfil a similar role as ample sets. It fundamentally differs from the ample set approach in that these assignments are not based on trace theoretical ideas nor seem to be reducible to such ideas.

5 Efficient Model Checking for a Subset of CTL

Model checking for CTL is linear in the size of the model acceptor and linear in the length of the tested formula [3]. However, as in the case of LTL, the number of states of the model acceptor is frequently exponential in the number of EN-system transitions. Moreover, also like for LTL, checking whether a CTL formula holds for an EN-system (i.e., it is true in its model) was proved to be $NP - hard$ in the number of transitions of the EN-system. It is therefore unlikely that a polynomial algorithm can be found.

In this section it is shown that the approach used to make model checking more efficient for LTL can be adapted to CTL. Equivalence robustness is redefined for CTL and reductions (trace automata) are redefined accordingly to respect this equivalence.

Again, formulas should not contain next step operators. The subset of CTL formulas satisfying this property is called CTL-x.

To start with, a standard method of CTL-x model checking is given, after which it is shown how the model acceptors it employs can be reduced using trace automata.

5.1 Standard Approach

Let P be an EN-system and $(\mathcal{T},\mathcal{I})$ be its trace semantics. Then, let $M_P = (F, V)$ be a model acceptor for the model M for $(\mathcal{T},\mathcal{I})$ and ψ be a given formula to be checked.

The method of CTL-x model checking is inductive, i.e., starting from the shortest and most deeply nested subformula φ of ψ the algorithm labels with φ these states of M_P, which accept traces at which φ holds in M. Therefore, in case of checking a one level less deeply nested subformula, it can be assumed that the states have just been labelled with all its subformulas. This process ends when ψ is attached to states; $M_P \models \psi$ iff ψ is attached to the root. Notice that the information about independent actions is not used.

Below, algorithms for labelling states of M_P are given.

Labelling p, $\neg\varphi$, $\varphi \wedge \gamma$

Notice, that:

1. $M_P, w \models q$ iff $V(w)(q) = true$, for $w \in PV$,

2. $M_P, w \models \neg\varphi$ iff not $M_P, w \models \varphi$,

   $M_P, w \models \varphi \wedge \gamma$ iff $M_P, w \models \varphi$ and $M_P, w \models \gamma$.

Therefore, in the first case it is checked whether $V(w)$ gives w the value $true$, in the second case it is checked whether w has been labelled φ and in the third case it is checked whether w has been labelled φ and γ.

Labelling $E(\varphi U \gamma)$

Observe that $M_P, w \models E(\varphi U \gamma)$ iff there is a state $v \in W$ and a sequence of states $w_0, \ldots, w_n \in W$ such that $w = w_0 \to \ldots \to w_n = v$ and $M_P, v \models \gamma$, $M_P, w_i \models \varphi$ for $0 \leq i < n$.

Firstly, all the states at which γ holds are labelled with $E(\varphi U \gamma)$. Secondly, the algorithm goes backward using the relation $\to^{-1}$ and labels all states at which φ holds with $E(\varphi U \gamma)$.

Labelling $EG\varphi$

Observe that $M_P, w \models EG\varphi$ iff there is a maximal path x starting at w such that $M_P, v \models \varphi$, for each v on x.

Let $W_\varphi = \{v \in W \mid M_P, v \models \varphi\}$ be the subset of W, labelled φ. A *strongly connected component* in W_φ is any subset $U \subseteq W_\varphi$ satisfying one of the following conditions:

- $(\forall v, v' \in U)$: $v \to^* v'$ and $v' \to^* v$, or
- U contains only one state that does not have any successor in W.

All the maximal strongly connected components in W_φ are selected. Notice that they are disjoint. Then, w is labelled $EG\varphi$ iff w is labelled φ and there is a maximal strongly connected component U in W_φ reachable from w by a path contained in W_φ.

5.2 Trace Approach

The idea is to use special kind of reduced model acceptors such they preserve all CTL-x formulas. First, it is shown which conditions should be satisfied by two model acceptors ensuring that the accepted models are CTL-x equivalent.

Let M and M' be two finite model acceptors.

Definition 5.1 ([2]) *A relation* $\sim \subseteq W \times W'$ *is a* stuttering equivalence *between* M *and* M' *if the following conditions hold:*

1. $w_0 \sim w'_0$,
2. *if* $w \sim w'$, *then* $V(w) = V'(w')$ *and for every maximal path* π *of* M *that starts at* w, *there is a maximal path* π' *in* M' *that starts at* w', *a partition* $B_1, B_2 \ldots$ *of* π, *and a partition* $B'_1, B'_2 \ldots$ *of* π' *such that for all* $j \geq 0$, B_j *and* B'_j *are nonempty and finite, and every state in* B_j *is related by '$\sim$' to every state in* B'_j *and*
3. *the same condition as (2) interchanging* π *and* M *with* π' *and* M'.

M and M' are said to be stuttering equivalent, *if there is a stuttering equivalence relation between them.*

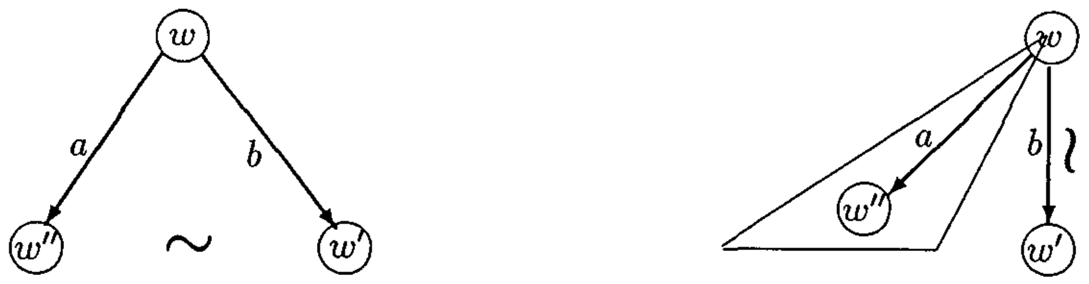


Figure 17: Possibilities of reduction by not expanding a.

In [2] stuttering equivalence is defined using approximants $\sim^n$. Because model acceptors are finite, it is easy to see that the two definitions are equivalent.

In order to avoid frequent referring to the model accepted by a model acceptor, the following convention is assumed : a formula φ is said to hold in a model acceptor M' (denoted $M' \models \varphi$) if φ holds in the model accepted by M'. A model checker enables to decide this fact automatically for a formula and an acceptor.

Theorem 5.2 ([2]) *Let φ be a CTL*-x formula. Let M and M' be two stuttering equivalent model acceptors. Then, $M \models \varphi$ iff $M' \models \varphi$.*

The next task is to generate a smaller model acceptor which is stuttering equivalent to the model acceptor M_P without first building M_P.

5.3 The Ample Set Algorithm

In the Section 4 two algorithms are given that generate trace automata preserving LTL-x formulas interpreted over observations. Presently, it is only known how to modify the ample set algorithm (see Figure 12) to generate automata preserving CTL-x formulas. The following exposition is based on [11].

The idea is to sharpen the requirements on generated subsets of successor states without changing the original ample set algorithm. Let Vis denote the set of *visible actions* in Σ, i.e., $Vis = \bigcup_{q \in PV} Vis(q)$. Note that if the formulas that need to be checked are known, then in order to get better reductions one should take PV to be the propositions occuring in these formulas only. The actions of $\Sigma \setminus Vis$ are called *invisible.*

To explain the restrictions imposed on the set *Ample*(w) let's assume first that the full model acceptor M_P does not contain loops except for self loops. Definition 5.1 describes the cases in which the model generator M' resulting after removing a transition from M_P is stuttering equivalent with M_P. In Figure 17 we have indicated the two situations in which the a-labeled transition need not be expanded from state w in M. This is the case when the states w'' and w' are stuttering equivalent (denoted $w'' \sim w'$) or when the states w and w' are stuttering equivalent ($w \sim w'$) in the full model M.

If $w'' \sim w'$ or $w \sim w'$, then for each path π with the prefix w, w'', there is a path π' with the prefix w, w' such that two partitions of π and π' satisfying Condition (2) of Definition 5.1 exist. Notice that it is because of the absence of non-trivial self-loops in M that the path π' cannot contain the transition $w \xrightarrow{a} w''$. If it could, M would not need to be stuttering equivalent with M' since $w \xrightarrow{a} w''$ is not present in M'. Therefore, the full model M remains stuttering equivalent with the model M' obtained after removing the transition $w \xrightarrow{a} w''$ from M.

As for ensuring $w'' \sim w'$, no efficiently checkable condition is known that would imply this. Indeed, the general problem is PSPACE-hard in the number of the transitions of EN-system, as it depends on the subgraphs of nodes reachable from w'' and w'. Therefore, we concentrate on the second case: ensuring that $w \sim w'$.

First observe that by repeatedly applying the argument above, it follows that if $w \sim w'$ then it suffices to only have the transition $w \xrightarrow{b} w'$ from w in M' and ignore the subtree of other transitions (indicated by the triangle in the figure). Hence, to reduce most effectively, we concentrate on singleton sets and impose the following constraint:

C0 *Ample*(w) contains either all actions enabled in state w , or exactly one of these; i.e., *Ample*(w) is a singleton.

The next condition will make sure that the execution of b does not change any propositional variable used in φ assigned to w and w', which is a necessary condition for $w \sim w'$.

In keeping with extant literature [32], this condition is called **C2**:

C2 If *Ample*(w) $\neq$ *en*(w), the action in *Ample*(w) is not visible.

In formulating the subsequent conditions we use the fact that we have already imposed conditions **C0** and **C2**.

The general problem of showing that $w \sim w'$ still is PSPACE-hard in the number of EN-system transitions. So, we aim for a stronger condition: for every path π starting in w there is a path π' starting in w' that is the same up to invisible actions. Now, consider π. As long as the actions along π are independent of b there is no problem in constructing π' because independent actions commute, so that these π-actions can still occur after the b-action. Dependent actions do cause a problem because there is no way to ensure that such actions can still occur without exploring all paths starting in w'. So, we disallow this situation by stipulating that such actions can only occur *after* the b-action has occurred.

C1 No action $a \in \Sigma \setminus$ *Ample*(w) that is dependent on the action in *Ample*(w) can be executed in P before the action from *Ample*(w) is executed, i.e., *Ample*(w) is a faithful set.

Now, consider the first action c along π that depends on b and let $\bar{w}$ be the state on π from which c is taken. Since the b-action must have occured along π before reaching

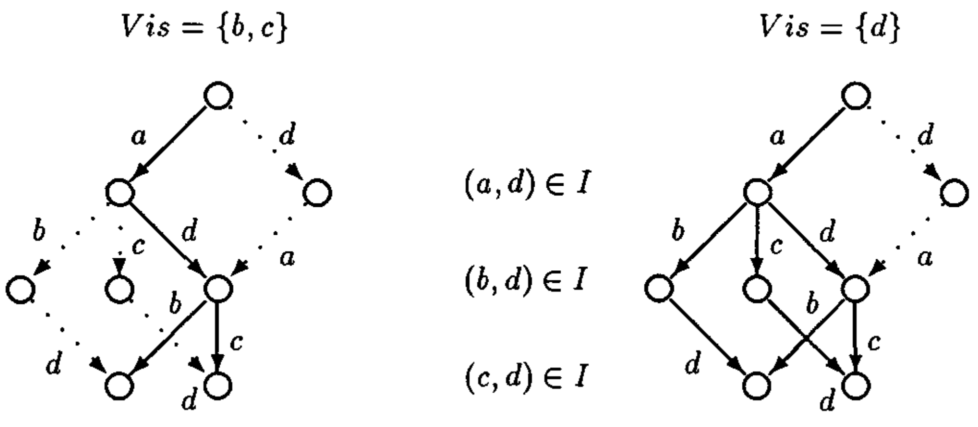


Figure 18: Correct CTL*-x reductions

state $\bar{w}$, commutativety of independent actions implies that the constructed prefix of π' ends in state $\bar{w}$, from which c and, indeed, the whole sequence of subsequent actions along π can be taken.

Condition **C1** occurs in many variations in LTL-x preserving reduction methods [18, 31, 45, 13]. The conditions **C1**, **C2** are sufficient to guarantee that the reduced state graph will preserve any checked *linear* temporal logic property φ [32]. The condition **C0** is sufficient for branching temporal logics.

An example of a reduction obtained by the algorithm using $Ample(w)$ satisfying conditions **C0**, **C1**, and **C2** is shown in Figure 18, where the dotted arrows correspond to transitions that are not expanded.

In the former section about efficient LTL model checking, as ample sets faithful ones were taken. However, this guaranteed only correct reductions for the concurrency fair version of LTL-x.

In Figure 19, the branching point after the execution of the visible action a is omitted in the reduction, which causes a distinction between the truth value of a CTL-x property in the full model acceptor (left) in which it does not hold and the reduced model acceptor (right) in which it holds.

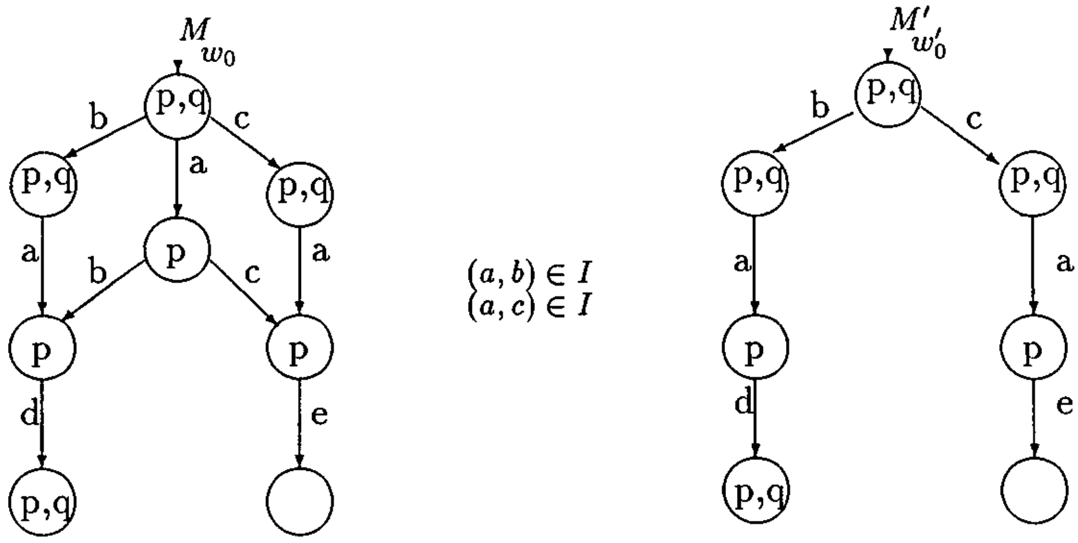
Let $M' = (F', V')$ be the model acceptor such that F' is a trace automaton on runs generated by the ample set algorithm under the assumption that $Ample(w)$ satisfies $\mathbf{C0} - \mathbf{C2}$, and $V' = V|W'$.

Theorem 5.3 *M_P and M' are stuttering equivalent.*

Proof: See [11] □

The above theorem together with Theorem 5.2 guarantees that the result of checking any CTL-x formula φ over M_P is the same as checking φ over M'.

Several experimental results showing how substantial reductions can be obtained by the ample set algorithm for LTL-x and CTL-x are shown in [11].



Let $\varphi = AG((p \wedge \neg q) \rightarrow (AFq \vee AF\neg p))$. Notice that $M', w'_0 \models \varphi$, but $M, w_0 \not\models \varphi$.

Figure 19: A reduction under **C1** and **C2** but not **C0** that does not preserve CTL-x

6 Model Checking for CTL$_P$

The logic CTL$_P$ has been introduced in order to express and prove partial order properties by model checking. Let's start the presentation with giving definitions of decidability and model checking for a logic.

Decidability of a logic concerns the existence of an algorithm to determine satisfiability of formulas. Such algorithm should decide the question: given a formula φ, is there a model M such that φ holds in M ? Model checking concerns the existence of an algorithm to determine satisfaction of formulas in given models. Such algorithm should decide the question: given a formula φ and a finite model acceptor M', does φ hold in the model M accepted by M' ? In Section 4, it has been shown that if a logic has a decision procedure of a certain form, namely using a finite structure representing all models, a model checking algorithm can be obtained in a standard manner.

Unfortunately, CTL$_P$ turns out to be not decidable, so this route could not be followed here. However, CTL$_P$ nevertheless does admit of a model checking algorithm. In this section undecidability of CTL$_P$ is shown and a model checking algorithm is presented.

The present exposition is based on the paper by Penczek [39].

6.1 Undecidability of CTL$_P$

Undecidability of CTL$_P$ is proved using the fact that the *recurring tiling problem* is known to be undecidable (see [15]). This problem concerns the existence of a certain covering with patterned tiles of the grid $\omega \times \omega$.

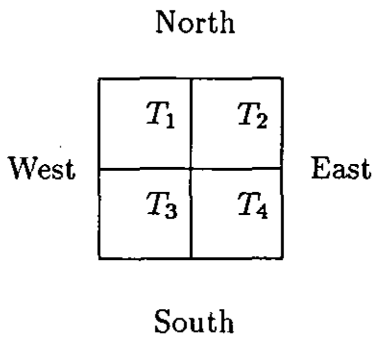


Figure 20: A compound tile c

It will be shown that the grid can be encoded as a CTL$_P$ frame and that the covering can be encoded as a CTL$_P$ formula. The recurring tiling problem then reduces to deciding the satisfiability of that formula.

To start with, the *recurring tiling problem* is stated. Let Γ be a finite set of types of tiles such that for each $T \in \Gamma$ each side - North, East, South and West - is assigned a number (N(T), E(T), S(T), and W(T), resp.). Let $Co \subseteq \Gamma^4$ be a set of *compound tiles*, i.e., big tiles, consisting of four tiles being put together as follows: $c = (T_1, T_2, T_3, T_4) \in Co$ iff $S(T_1) = N(T_3)$, $E(T_1) = W(T_2)$, $N(T_4) = S(T_2)$, and $W(T_4) = E(T_3)$ (see Figure 20).

The idea is now to try to cover each lattice point with a compound tile in such a manner that a certain pattern ensues.

For each $c = (T_1, T_2, T_3, T_4) \in Co$, let $U(c), R(c) \subseteq Co$, where U stands for *Up* and R stands for *Right*, with $U(c) = \{c' \in Co \mid c' = (T, T', T_1, T_2)$, for some $T, T'\}$, $R(c) = \{c' \in Co \mid c' = (T_2, T, T_4, T')$, for some $T, T'\}$. Let $T_0, T_f \in \Gamma$ be two special types. The problem is to find a compound tile assignment $c : \omega \times \omega \longrightarrow Co$ such that for all $i, j \in \omega$, $c(i,j) \in Co$, $c(0,0) \in \{c' \in Co \mid c' = (T, T', T_0, T'')$, for some $T, T', T''\}$, $c(i, j+1) \in U(c(i,j))$, and $c(i+1, j) \in R(c(i,j))$, and there are infinitely many compound tiles in the leftmost column, which contain the tile type T_f. The above formulation means that one has to find a compound tile assignment of the lattice points in the plane such that if a point is assigned a compound tile c, then the point just above is assigned a compound tile from the set $U(c)$ and the point to the right is assigned a compound tile from the set $R(c)$, the beginning is assigned a compound tile of a given subset of Co and the given tile type T_f occurs infinitely often in the leftmost column (see Figure 21).

Next, it is shown how to reduce the recurrent tiling problem to the satisfiability of a CTL$_P$ formula. This is possible since CTL$_P$ formulas can characterize frames

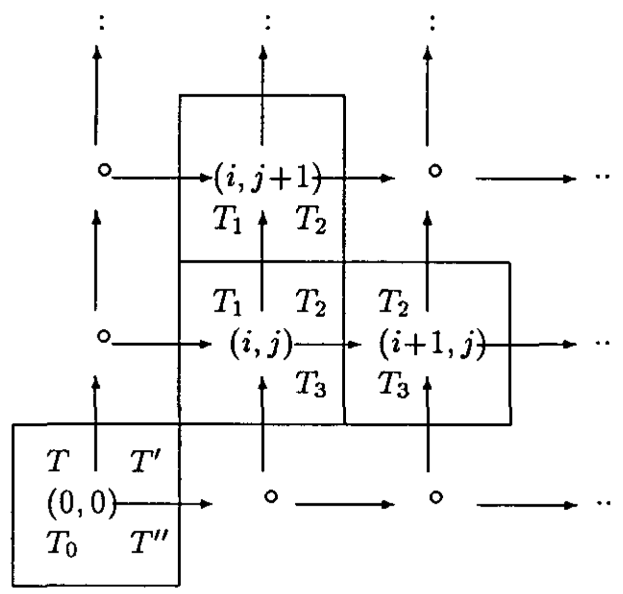


Figure 21: The patterning requirements of a covering with compound tiles

(and models) of trace systems up to isomorphism (see Section 3.6).

To enable the encoding of the tiling problem into CTL$_P$, a frame for CTL$_P$ that is isomorphic with the grid $\omega \times \omega$ is defined.

Let $\Sigma = \{a, b\}$ and $I = \{(a, b), (b, a)\}$. Consider the trace system $\mathcal{T} = [\Sigma^*]$. The frame for $\mathcal{T}$ is defined as $F = (\{w_{i,j} \mid (i,j) \in \omega \times \omega\}, \Sigma, \rightarrow, w_{0,0}\}$, where $w_{i,j} \xrightarrow{a} w_{i',j'}$ iff $i' = i + 1$ and $j' = j$ and $w_{i,j} \xrightarrow{b} w_{i',j'}$ iff $i' = i$ and $j' = j + 1$

F is obviously isomorphic (up to the labels of transitions) with the grid $\omega \times \omega$ (see Figure 22).

Let $PV = \{C_i \mid c_i \in Co\} \cup \{p_a, p_b\}$, i.e., each compound tile c_i is assigned the atomic proposition C_i. The atomic propositions p_a and p_b are defined to encode the labels of transitions, which allows to define the labelled next step operators (see Section 3.2).

Now it is possible to give a set of formulas of CTL$_P$ such that its conjunction is satisfiable iff the recurring tiling problem has a solution.

The following CTL$_P$ formulas characterize F up to isomorphism.

(A):

1. $AG(EX_a true \wedge EX_b true)$

2. $EF(EY_a true \wedge EY_b true)$

1) expresses that two actions a and b are executed at each state. 2) says that a and b are independent at some state. Thus, by the definition of independence relation, a and b are independent at each state.

$$\begin{tikzcd}
\vdots & \vdots & \vdots & \\
w_{0,2} \arrow[u, "b"] \arrow[r, "a"'] & w_{1,2} \arrow[u, "b"] \arrow[r, "a"'] & w_{2,2} \arrow[u, "b"] \arrow[r, "a"'] & \cdots \\
w_{0,1} \arrow[u, "b"] \arrow[r, "a"'] & w_{1,1} \arrow[u, "b"] \arrow[r, "a"'] & w_{2,1} \arrow[u, "b"] \arrow[r, "a"'] & \cdots \\
w_{0,0} \arrow[u, "b"] \arrow[r, "a"'] & w_{1,0} \arrow[u, "b"] \arrow[r, "a"'] & w_{2,0} \arrow[u, "b"] \arrow[r, "a"'] & \cdots
\end{tikzcd}$$

Figure 22: The frame F isomorphic with the grid $\omega \times \omega$

The following formulas describe the tiling.
(B):

1. $\bigvee\{C_i \mid c_i = (T, T', T_0, T''),$ for some $T, T', T''\}$,
2. $AG(\bigvee(\{C_i \mid c_i \in Co\}) \wedge (\bigwedge\{C_i \rightarrow \neg C_j \mid i \neq j\}))$,
3. $AG(\bigwedge(\{C_i \rightarrow EX_a(\bigvee\{C_j \mid c_j \in R(c_i)\}) \mid c_i \in Co\}))$,
4. $AG(\bigwedge(\{C_i \rightarrow EX_b(\bigvee\{C_j \mid c_j \in U(c_i)\}) \mid c_i \in Co\}))$,
5. $AG(\neg EY_a true \rightarrow EF(\neg EY_a true \wedge \bigvee\{C_i \mid$ type T_f occurs in compound tile $c_i\}))$.

1) expresses that the tile type T at the beginning is T_0. 2) enforces exactly one compound tile at each point of the grid. 3) and 4) ensure that successors have the right compound tile. 5) requires that the tile type T_f occurs infinitely often in the leftmost column.

The recurring tiling problem has a solution iff the conjunction of the formulas A) and B) is satisfiable. Therefore, the validity problem for CTL$_P$ is undecidable.

In fact, using the same method, one can show that if labelled next step operators can be encoded in the language of ISTL, then ISTL is not decidable as well [37].

6.2 NP-hardness of CTL$_P$ Model Checking

The model checking problem for CTL$_P$ is formulated as: given a formula φ and given a finite model acceptor M', does φ hold in the model M accepted by M' ?

NP-hardness of CTL$_P$ model checking problem follows intuitively from the fact that the logic allows to express properties about selected states of a path (by using past operators over models which look like tree). Model checking problem for CTL is of polynomial complexity, but CTL allows only for saying something about all states or some state of a path.

Then, NP-hardness of CTL$_P$ model checking is proved using the fact that the *3-SAT problem* is NP-hard. This problem concerns the decidability of certain propositional formulas, i.e., whether or not a model exists for which such formula holds.

It will be shown that this problem can be encoded as checking whether or not some CTL$_P$ formula holds in some model and that the encoding is polynomial (in fact, even linear).

Note that here a decidability problem in one framework is reduced to a model checking problem in another framework.

It is proved that 3-SAT problem is polynomially reducible to determining whether $M \models \varphi$, for some CTL$_P$ formula φ and some model acceptor M' of M.

To start with, the 3-SAT problem is stated. Let $\psi = b_1 \wedge \ldots \wedge b_m$ be a boolean formula in 3-CNF, i.e., $b_i = l_{i_1} \vee l_{i_2} \vee l_{i_3}$, for $1 \leq i \leq m$, $l_{i_j} = x_k$ or $\neg x_k$ for some k such that $1 \leq k \leq n$, where $x_1, \ldots, x_n$ are the propositional variables appearing in ψ.

Next, it is shown how to polynomially reduce the 3-SAT problem to a CTL$_P$ model checking problem.

Let $M' = (F', V')$ be a model acceptor (see Figure 23), defined as follows:

- Let $X = \{x_i \mid 1 \leq i \leq n\} \cup \{x'_i \mid 1 \leq i \leq n\}$, $Y = \{y_i \mid 0 \leq i \leq n\}$, where $X \cap Y = \emptyset$,
- $F' = (W', \Sigma, \rightarrow', I, w'_0)$, where
- $W' = X \cup Y$, $\Sigma = X \times Y \cup Y \times X$, $I = \emptyset$,
- $\rightarrow' = \{(y_{i-1}, (y_{i-1}, x_i), x_i), (y_{i-1}, (y_{i-1}, x'_i), x'_i), (x_i, (x_i, y_i), y_i), (x'_i, (x'_i, y_i), y_i) \mid 1 \leq i \leq n\}$,
- $w'_0 = y_0$,
- $PV = \{b_i \mid 1 \leq i \leq m\}$,
- $V'(x_i)(b_j) = true$ iff x_i appears as a literal in b_j,
- $V'(x'_i)(b_j) = true$ iff $\neg x_i$ appears as a literal in b_j,
- $V'(y_i)(b_j) = false$, for $0 \leq i \leq n$ and $0 \leq j \leq m$.

The following equivalence holds:

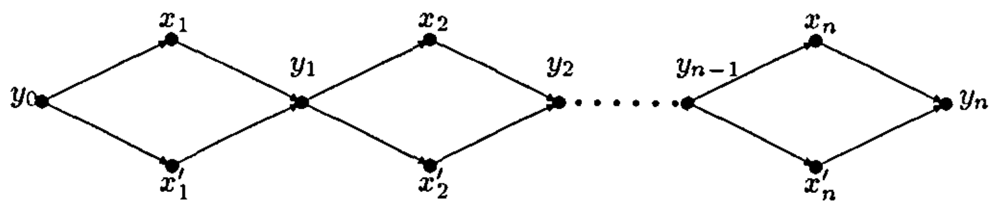


Figure 23: The rats F' of the model acceptor M'

*) ψ is satisfiable iff $M \models EF(EPb_1 \wedge \ldots \wedge EPb_m)$,
where M is the model accepted by M'.

Each path represents one of the possible valuations of the propositional variables x_i. Since the independence relation $I = \emptyset$, the model accepted by M' is a tree. Therefore, to satisfy ψ, there must be one such path on which each b_i is true somewhere. As the number of states in the model acceptor is linear in the number of propositions in the formula, this reduction is linear. Hence, model checking for CTL$_P$ is NP-hard.

6.3 Model Checking for CTL$_{P-}$

To ease the exposition, initially a restricted version of the logic, called CTL$_{P-}$, is considered which contains only CTL$_P$ formulas without nested past modalities.

Although CTL$_{P-}$ seems to be a small extension of CTL, it turns out that all the partial order properties discussed so far are expressible in CTL$_{P-}$. Therefore it is easy to see that the proofs of undecidability and of NP-hardness of model checking also hold for CTL$_{P-}$.

Because of NP-hardness, only a one exponential model checking algorithm is presented for the restricted logic. It is shown at the end of the section that the algorithm can be extended for CTL$_P$ model checking.

Unfortunately, neither of the two known methods for model checking applied to CTL* (LTL) or CTL, can be used directly in the case of CTL$_{P-}$.

The first known method, for CTL* model checking [8] or as encountered in Section 4 for LTL model checking [20, 47], requires building a model structure for a formula. It follows from the undecidability of CTL$_P$ that this is not possible now.

The second known method, as encountered in Section 5, is for CTL [3]. The idea there is to iteratively label with progressively more complicated subformulas of the formula (which is checked) the states of M' where these subformulas hold. Showing where this methods fails in the present setting is used as an introduction to the method presented here.

The problem is caused by the combination of past operators and partial order semantics. As stated in Section 2.6, the requirements on an equivalence relation used to construct a model acceptor are just that traces from one equivalence class can neither be distinguished by their interpretation nor by their continuations. As CTL formulas are only concerned with the future, a CTL formula then has the same value on each trace of such class - which enables to decide whether or not the formula should be added as a label.

The equivalence relation provides no such property for the prefixes of the traces from one equivalence class. Indeed, as will be shown, CTL$_{P-}$ formulas with past time operators turn out to potentially evaluate differently on different traces from one class - which makes it no longer possible to decide about adding such formulas as labels. The solution is of course to sharpen the equivalence relation accordingly.

Contrary to what naively might be expected, this is not a simple extension of the ideas from CTL. Herein lies the novelty of the approach presented as well as the connection with trace theory.

To ease the understanding, it is first shown how the requirements on the equivalence relation defined on the model need to be strengthened to accommodate CTL$_{P-}$ and next how a model acceptor that satisfies these requirements can be constructed directly from a given model acceptor, i.e., without referring to the model.

Let $M' = (F', V')$ be the model acceptor of a model M built using an equivalence relation as defined in Definition 2.19. There are two cases to be investigated, corresponding to the past time operators EP and EY.

> Consider two traces $[abc]$ and $[def]$ such that they belong to the same equivalence class and $M, [abc] \models p$ (then also $M, [def] \models p$). All actions are interdependent.

1. Let $M, [\epsilon] \models p$, $M, [a] \models p$, $M, [ab] \models p$, and $M, [d] \models \neg p$, $M, [de] \models p$.

   Then, EPp evaluates differently on $M, [abc]$ than on $M, [def]$, but EYp evaluates to the same value on both.

2. Now, let $M, [\epsilon] \models p$, $M, [a] \models p$, $M, [ab] \models \neg p$, and $M, [d] \models \neg p$, $M, [de] \models p$.

   Then, EYp evaluates differently on $M, [abc]$ than on $M, [def]$, but EPp evaluates to the same value on both.

To achieve that such formulas do not evaluate differently on one equivalence class, the original equivalence relation is strengthened, by the following extra requirements.

Let M'' be a new model acceptor, defined using an equivalence relation satisfying the conditions as before plus the requirement that its acceptance function AC'' satisfies the following properties concerning the prefixes of traces, defined in terms of AC': $AC''(\tau) = AC''(\tau')$ implies

P1) $AC'(\downarrow \tau) = AC'(\downarrow \tau')$, and

P2) for each $a \in \Sigma$, if $\tau = \tau_1[a]$, then $\tau' = \tau_1'[a]$ and $AC'(\tau_1) = AC'(\tau_1')$, for $\tau_1, \tau_1' \in [\Sigma^*]$.

P1) ensures that traces accepted by the same state in M'' satisfy the same EP formulas, whereas P2) ensures the same for EY formulas. In fact, AC'' ensures that each CTL$_{P-}$ formula receives just one value on all traces accepted at the same state.

The next task is to construct M'' directly from M' rather than define M'' via the model M. To see how this can be done, an equivalence relation EQ that provides an AC'' with the above properties is defined, again, firstly on the states $Tr(F')$ of the model M.

Let $P(\Sigma)$ be the family of all subsets of Σ and *one* be the following function $one : Tr(F') \longrightarrow \{tt, ff\}$ such that $one(\tau) = tt$, if $|\tau| = 1$, and $one(\tau) = ff$, if $|\tau| \neq 1$, where (tt stands for *true* and ff stands for *false*).

Now a function $Rep : Tr(F') \longrightarrow 2^{W' \times P(\Sigma) \times \{tt,ff\}}$ is defined such that $Rep(\tau) = \{(AC'(\tau_1), \Sigma(\tau_2), one(\tau_2)) \mid \tau_1\tau_2 = \tau\}$. The intuitive meaning is as follows. $AC'(\tau_1)$ records states of W' accepting the prefixes of τ. $\Sigma(\tau_2)$ records names of actions which occur in the trace leading from the prefix to τ. $one(\tau_2)$ says whether that trace contained one or more actions.

Then, using *Rep*, the equivalence relation EQ and its equivalence classes are defined.

- $\tau \; EQ \; \tau'$ iff $Rep(\tau) = Rep(\tau')$, for $\tau, \tau' \in Tr(F')$
- $[\tau]_{EQ} = \{\tau' \in Tr(F') \mid \tau \; EQ \; \tau'\}$,

Let $F'' = (W'', \Sigma, \rightarrow'', I, w_0'')$ be the quotient structure of $Tr(F')$ by the equivalence relation EQ such that the elements of W'' are of the form $Rep(\tau)$ for $\tau \in Tr(F')$ rather than $[\tau]_{EQ}$. It is shown in [39] that the acceptance function AC'' for F'' satisfies the properties P1) and P2).

It is now shown that F'' can be defined directly from F'. The following notions simplify the rest of the construction.

- a function $last : W'' \longrightarrow W'$, such that $last(w'') = w'$, if $(w', \emptyset, ff) \in w''$, $last(w'')$ gives the state of M' accepting the traces accepted by w'' in M'',
- a function $emp : P(\Sigma) \longrightarrow \{tt, ff\}$ such that $emp(\Delta) = tt$, if $\Delta = \emptyset$, and $emp(\Delta) = ff$, if $\Delta \neq \emptyset$.

The following partial function $Next : W'' \times \Sigma \longrightarrow W''$ allows to construct for a given $w'' \in W''$ and $a \in en(w'')$ the state reached in W'' after executing a at w'', i.e., $w'' \overset{a}{\rightarrow}'' Next(w'', a)$ holds.

- $Next$ is defined as: $Next(w'', a) = \{(w', \Delta \cup \{a\}, emp(\Delta)) \mid (w', \Delta, q) \in w''\} \cup \{(w_a', \Delta, q) \mid (w', \Delta, q) \in w''$ and $\{a\} \times \Delta \subseteq I\}$, for any $w'' \in W''$ and $a \in en(last(w''))$, where w_a' denotes the state in W' reached after executing a at w'.

The correctness of the definition of $Next$ follows from the following special case of Levi's Lemma for traces [24].

Lemma 6.1 *For each $\tau, \tau_1, \tau_2 \in [\Sigma^*]$ and $\alpha \in [\Sigma]$ the following condition holds:*

$\tau\alpha = \tau_1\tau_2$ *iff*

1) $\tau_2 = \tau''\alpha$ and $\tau_1\tau'' = \tau$, for some $\tau'' \in [\Sigma^*]$ or

2) $\tau_1 = \tau'\alpha$, $\{a\} \times \Sigma(\tau_2) \subseteq I$, and $\tau'\tau_2 = \tau$, for some $\tau' \in [\Sigma^*]$ and $[a] = \alpha$.

This lemma allows to represent $Next(w'', a)$ as the union of two sets corresponding to the conditions 1) and 2).

Construction of F''

The construction of F'' is performed now inductively, in stages.

Let $F''_1 = (W''_1, \Sigma, \emptyset, I, w''_0)$, where $W''_1 = \{w''_0\}$ and $w''_0 = \{(w'_0, \emptyset, ff)\}$. Now, for each node w'' of $W''_i - W''_{i-1}$ (with $W''_0 = \emptyset$) the algorithm adds all its successors in W'' and extends $\rightarrow''_i$, respectively. Let $F''_i = (W''_i, \Sigma, \rightarrow''_i, I, w''_0)$ and $W^{\emptyset}_i = W''_i - W''_{i-1}$. Then, $F''_{i+1} = (W''_{i+1}, \Sigma, \rightarrow''_{i+1}, I, w''_0)$, where

- $W''_{i+1} = W''_i \cup \{Next(w'', a) \mid w'' \in W^{\emptyset}_i,\ a \in en(last(w''))\}$,
- $\rightarrow''_{i+1} = \rightarrow''_i \cup \{(w'', a, Next(w'', a)) \mid w'' \in W^{\emptyset}_i,\ a \in en(last(w''))\}$.

The construction stops at the least integer m with $F''_m = F''_{m+1}$. It is easy to see that F''_m is isomorphic to F''.
Then, the new model acceptor $M'' = (F'', V'')$ is defined as follows:

- $F'' = (W'', \Sigma, \rightarrow'', I, w''_0)$,
- $V''(w'')(q) = V'(last(w''))(q)$, for $q \in PV$.

The complexity of building M'' is: $O(|\rightarrow'| \times |P(\Sigma)| \times 2^{2 \times |W'| \times |P(\Sigma)|})$.
Notice that for each subformula φ of CTL$_{P-}$ formula ψ and for each $\tau, \tau' \in Tr(F'')$, if $AC''(\tau) = AC''(\tau')$, then $M, \tau \models \varphi$ iff $M, \tau' \models \varphi$. Therefore, $w'' \in W''$ is labelled φ (written $M'', w'' \models \varphi$), if $M, \tau \models \varphi$, for some τ with $AC''(\tau) = w'$.

Obviously, for each subformula φ without past operators, for each $\tau, \tau' \in Tr(F')$, if $AC'(\tau) = AC'(\tau')$, then $M, \tau \models \varphi$ iff $M, \tau' \models \varphi$. Therefore, $w' \in W'$ is labelled φ (written $M', w' \models \varphi$), if $M, \tau \models \varphi$, for some τ with $AC'(\tau) = w'$.

Next, algorithms labelling the states of M'' are shown. The method is inductive, i.e., given a formula ψ, starting from its shortest and most deeply nested subformula φ the algorithm labels with φ these states of M'', which accept traces, at which φ holds in M. Therefore, in case of checking a less nested subformula, it can be assumed that the states have just been labelled with all its subformulas.

Firstly, the states of M' are labelled with all the subformulas of ψ which do not contain past subformulas. Then, the states of M'' are labelled with all the subformulas of ψ. Below, algorithms for labelling states of M'' are given.

Labelling p, $\neg\varphi$, $\varphi \wedge \gamma$, $E(\varphi U \gamma)$

The algorithms have been shown in Section 5.

Labelling $EY\varphi$

Observe that $M'', w'' \models EY\varphi$ iff there is $(w', \Delta, tt) \in w''$ such that $M', w' \models \varphi$.

Labelling $EP\varphi$

Observe that $M'', w'' \models EP\varphi$ iff there is $(w', \Delta, q) \in w''$, such that $M', w' \models \varphi$.

Labelling $EX\varphi$

Observe that $M'', w'' \models EX\varphi$ iff there is $v'' \in W''$ such that $w'' \rightarrow'' v''$ and $M'', v'' \models \varphi$. Therefore, the algorithm finds all the states at which φ holds and labels all its predecessors with $EX\varphi$.

Labelling $EG\varphi$

Observe that $M'', w'' \models EG\varphi$ iff there is an observation x starting at w'' such that $M'', v'' \models \varphi$, for each v'' on x.

Let $W''_\varphi = \{v'' \in W'' \mid M'', v'' \models \varphi\}$ be the subset of W'', labelled φ. A *strongly connected component* in W''_φ is any subset $U \subseteq W''_\varphi$ satisfying one of the following conditions:

- $(\forall v, v' \in U)$: $v \rightarrow''^* v'$ and $v' \rightarrow''^* v$, or
- U contains only one state that does not have any successor in W''.

Secondly, all the maximal strongly connected components in W''_φ are selected. Notice that they are disjoint. Then, w'' is labelled $EG\varphi$ iff w'' is labelled φ and there is a maximal strongly connected component U in W''_φ reachable from w'' by a path contained in W''_φ and for all $a \in \Sigma$ there is v in U such that a is not enabled at v or a is dependent with at least one action enabled at v and leading to a state from U (i.e., U "contains" an observation).

6.3.1 Complexity of CTL$_{P-}$ Model Checking

In order to handle an arbitrary CTL$_{P-}$ formula ψ, the state-labelling algorithm is applied to the subformulas of ψ starting with the shortest and most deeply nested one. Since ψ contains at most $length(\psi)$ different subformulas, the algorithm requires time

$$O(length(\psi) \times \mid \rightarrow' \mid \times |P(\Sigma)| \times 2^{2 \times |W'| \times |P(\Sigma)|}).$$

In [39], several improvements are defined to the algorithm that decrease its complexity.

Example 6.2 [proving serializability] Below, it is shown how to prove serializability of the trace system $\mathcal{T}_2$ of Example 2.5.

It is to be verified:

1. $M_2 \models p_1 \wedge p_2 \wedge p_9 \rightarrow AFEP((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9))$,
2. $M_2 \models AG[((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9)) \rightarrow AF(EP(p_7 \wedge p_8 \wedge p_9)]$.

Since M_2 is finite, it is taken as its model acceptor M_2'. Moreover, observe that in this case M_2'' is equal to M_2'. Therefore, the states of M_2 are labelled. The only difficult task is to label the states of M_2 with the formulas:

$$F1 = AFEP((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9)),$$

$$F2 = AFEP(p_7 \wedge p_8 \wedge p_9).$$

It is shown how to do that for the formula F1. First notice that:
$F1 \equiv \neg EG \neg EP\varphi$, where $\varphi = ((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9)$. Next, the states of M_2 are labelled with $EG \neg EP\varphi$.

- $M_2, w \models \varphi$, if $w \simeq [a_1a_2a_3]$ or $w \simeq [b_1b_2b_3]$.
- $M_2, w \models EP\varphi$ for all w such that $w \simeq \tau$ for $\tau \in \uparrow [a_1a_2a_3] \cup \uparrow [b_1b_2b_3]$.
- M_2, w_0 is not labelled $EG \neg EP\varphi$.

Thus, $M_2, w_0 \models AFEP((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9))$. Consequently, $M_2, w_0 \models p_1 \wedge p_2 \wedge p_9 \rightarrow AFEP((p_7 \wedge p_2 \wedge p_9) \vee (p_1 \wedge p_8 \wedge p_9))$.

6.3.2 Extending Model Checking to CTL$_P$

The presented method of model checking can be extended such that past formulas can be nested.

Define a sequence of unfoldings $M_1, \ldots, M_n$, where $M_1 = M''$, n is the maximal depth of nested past formulas in ψ, M_{i+1} is obtained from M_i in the same way as M'' was obtained from M'. Then, the states of M_i are labelled inductively with subformulas of ψ containing the nested past formulas of depth at most i. Then, $M, w_0 \models \psi$ iff the beginning state of M_n has been labelled with ψ. In the worst case the complexity is $exp_n(2 \times |W'| \times |P(\Sigma)|)$.

7 Proving Program Properties using ISTL$_P$

As discussed in the introduction to this chapter, trace systems are special in that they both enable very detailed modelling and also offer opportunities for abstraction. Interleaving Set Temporal Logic* (ISTL*), the propositional version of which was introduced in Section 3, enables to exploit both features. This leads to two separate approaches; an example of both will be given. The part concerning expressiveness is based on Peled and Pnueli's [34], the part concerning abstraction is based on Katz and Peled's [18].

The need for verification by hand is most obvious where automated verification is intrinsically impossible. This is foremost the case when properties of systems are considered that concern potentially unbounded data. Finite state descriptions no longer suffice there. Consequently, propositional logics, representing data values as propositions, do not apply. The ensuing extension to first order logic precludes automated verification.

To allow giving a trace semantics to programs with variables, trace systems are used that support state variables. Propositional ISTL is then extended accordingly to first order ISTL. Furthermore, past time modalities are added. The resulting logic is called ISTL$_P$. These extra modalities are, as will be seen, crucial for the proof system. Following that, programs are introduced and the connection between behaviour of programs and ISTL$_P$ formulas is made. Finally, a proof system, or, in fact, two proof systems, for the program part are presented, corresponding to the two different strengths of trace systems.

As the interesting features of the approach can already be demonstrated on simple finite state examples, only such are presented.

7.1 Programs

Like before, interpreted trace systems are used as the basis for program behaviour; interpretation is used in this case to accommodate state variables.

Because ISTL$_P$ is interpreted on models rather than on interpreted trace systems, the corresponding models will be taken as the semantics of programs.

Programs themselves are taken as quadruples consisting of an independence alphabet, an ordered set of variables and the initial condition. Programs are defined in a first order language $\mathcal{L}$ interpreted over a first order structure $\mathcal{A}$ for which the standard interpretation of the relation and function symbols is assumed.

An *interpretation* $\mathcal{J}$ of a vector of variables $\vec{y}$ is a mapping associating with each variable y from $\vec{y}$ a value $\mathcal{J}(y)$ from its domain. Let $|\vec{y}|$ denote the number of variables in $\vec{y}$.

Definition 7.1 *A* program *$\mathcal{P}$ is a quadruple $<\Sigma, I, \vec{y}, \Theta>$, where (Σ, I) is an independence alphabet, $\vec{y}$ a finite vector of program variables and Θ a first order formula with free variables from $\vec{y}$ (the* initial condition*). Furthermore, with each action a is associated a first order formula en_a (the* enabling condition*) and a $|\vec{y}|$-tuple of terms f_a (*transformation function*), satisfying the following consistency requirements for each $(a,b) \in I$:*

- $(en_a(\vec{y}) \wedge en_b(\vec{y}) \rightarrow f_a(f_b(\vec{y})) = f_b(f_a(\vec{y}))$
  (commutativity of independent actions),

- $en_a(\vec{y}) \rightarrow (en_b(\vec{y}) \leftrightarrow en_b(f_a(\vec{y})))$
  (independent actions do not influence each others enabledness).

The enabling condition en_a holds if the action a is enabled, whereas the transformation function f_a gives the new values to the variables of $\vec{y}$ after executing a.

The consistency requirements ensure that the effect of the actions on the variables is compatible with inverting the order of execution between adjacent independent actions.

To simplify the exposition, the programs are restricted such that Θ uniquely determines the value of $\vec{y}$. The extension to the unrestricted case is straightforward.

It is easy to see, especially considering their semantics, that EN-systems (see Definition 2.26) can be regarded as a special case of the programs defined above. Therefore the running examples can again be used, as these have been shown, in Section 2.3 to be expressible as EN-systems.

Semantics

The semantics of programs is defined in two steps: the interpreted trace system corresponding to a program is defined and then the program semantics is given as the model corresponding to that.

Definition 7.2 *The* interpreted trace system corresponding to a program $\mathcal{P} =< \Sigma, I, \vec{y}, \Theta >$, *is* $(\mathcal{T}, \mathcal{I})$ *where:*

- $\mathcal{T}$ *is the minimal set of traces over* (Σ, I) *containing* $[\varepsilon]$ *such that if* $\tau \in \mathcal{T}$ *and* $\mathcal{I}(\tau)$ *validates* en_a *(a is enabled at* τ*), then* $\tau[a] \in \mathcal{T}$,
- $\mathcal{I}[\varepsilon]$ *validates* Θ *and if* $\tau[a] \in \mathcal{T}$*, then* $\mathcal{I}(\tau[a]) = f_a(\mathcal{I}(\tau))$.

Note that the consistency conditions in the definition of program ensures that $\mathcal{I}$ is well defined.

Also note that the use of variables is quite different than in Section 3.4. The concern there was to code into the state precisely the information needed to capture transition labellings. The aim here is to capture for each action the effect on the state variables - this may or may not capture the transition labellings.

Definition 7.3 *The* semantics *of a program* $\mathcal{P} =< \Sigma, I, \vec{y}, \Theta >$ *is the model for the interpreted trace system* $(\mathcal{T}, \mathcal{I})$ *representing the behaviour of* $\mathcal{P}$.

This model is called the model for P.

7.2 First Order ISTL$_P$

The definition of ISTL is extended to its first order version with past operators.

Syntax of first order ISTL$_P$

Only two entries in the definition of ISTL, given in Section 3, change, as follows.

S1. if φ is a first order formula, then φ is a formula,

S3. if φ, ψ are formulas, then so are $EX\varphi$, $EG\varphi$, $E(\varphi U\psi)$, $EY\varphi$ and $EP\varphi$.

EY is the backward step modality and EP denotes sometime in the past. As in case of CTL$_P$, only two past modalities are added to the language of ISTL. Extensions with $EH\varphi$ and $E(\varphi S\psi)$ are possible, but will not be discussed in this chapter.

Semantics of first order ISTL$_P$

Because a first order version of ISTL$_P$ is considered, models are extended by the first order structure $\mathcal{A}$. Let (F,V) be the model for an interpreted trace system $(\mathcal{T},\mathcal{I})$, as defined in Section 3.1.5. Then $M = (\mathcal{A},F,V)$ is the extension to first order, i.e., for every world $x \in W$, $(\mathcal{A},V(x))$ is a first order model on which formulas without temporal operators can be interpreted.

Clause $S1$ below just states that then evaluation of non-temporal formulas is as usual. Let $F' = (W',\Sigma,\rightarrow',w_0)$ be a run of F. Clause $S3$ gives the semantics of the two new entries in the syntax, where $x \in W'$ and o is an observation of F'.

S1. $x \models \varphi$ iff $(\mathcal{A},V(x)) \models \varphi$, for non-temporal φ,

S3. $M,x \models EY\varphi$ iff $M,x' \models \varphi$, for some $x' \in W$ with $x' \rightarrow x$,

$M,x \models EP\varphi$ iff $M,x' \models \varphi$, for some $x' \in W$ with $x' \rightarrow^* x$.

As before, a first order ISTL$_P$ formula φ is *valid* in a model M (written $M \models_{ISTL_P} \varphi$) iff $(\mathcal{A},F',V'),w_0 \models \varphi$, for all runs F' of F, where $V' = V|W'$.

Definition 7.4 *A formula φ is* valid for a program $\mathcal{P}$ *if it is valid in the model $(\mathcal{A},F,V)$ for P.*

7.3 Proof Systems for Programs

When a *logic* is used to describe properties of *programs* that manipulate *data*, three different parts can be distinguished accordingly in the proof system:

- the logic part - to deal with logical truth,
- the (data) domain part - to deal with domain properties, and
- the program part - to deal with program properties.

The program part is used to remove the references to the program from formulas. The logic and domain part are then used to derive the truth of the resulting formulas.

Because of the incompleteness incurred generally for first order logics over interpreted domains, only relative completeness can be achieved. This means that applying the program part enables to reduce the proof obligation about the program to proving validity of formula, without further reference to the program. The validity of some of the resulting formulas then has to be given by an oracle. These formulas could be temporal formulas, expressing a complicated mixture of properties of the data domain and the trace structure. As the incompleteness is really due to the interpretation of the data domain in first order logic and the modal logic is introduced primarily to express properties of the interdependency of actions, the oracle should preferably be limited to first-order non-temporal formulas only. This result is achieved for ISTL$_P$, both in case of exploiting expressiveness as when exploiting abstraction.

Rather than giving full proof systems, the present exposition is limited to the relevant and novel aspects of the programming part of the two systems considered.

7.4 Proof System to Exploit Expressiveness

Linear Time Temporal Logic, as discussed in Section 3, uses sequences, i.e., paths, of states as models. ISTL* uses observations of runs. The small discrepancy between the set of paths and the set of observations generated by a program can be taken care of by requiring the appropriate condition explicitly by means of the LTL formula

$$Fair \;=\; \bigwedge_{a\in\Sigma}(FGinden_a \rightarrow GFex_a),$$

where $inden_a$ means that a is enabled and independent with the action which is executed, ex_a means that a is executed.

ISTL* can be viewed as extending Linear Time Temporal Logic in the sense that an LTL formula $Fair \rightarrow \varphi$ is equivalent to ISTL* formula $A\varphi$.

Considering proof systems, this means that ISTL formulas corresponding to LTL-ones can be handled as these. Program proof rules for LTL can be found in [22]. ISTL$_P$ formulas that exploit the extra expressiveness of ISTL$_P$, i.e., do not quantify over all observations but rather assert the existence of one, have no LTL analogon. At present, complete proof rules are only known for certain, important, classes of such formulas, namely, those of form

$$AG(\varphi \rightarrow EX\psi)$$

and

$$AG(\varphi \rightarrow E(\phi U\psi))$$

as well as their past time counterparts.

A general strategy to prove eventuality properties is firstly to prove properties that hold after one step and then, secondly, to combine these results. The crucial

/

proof principles for this approach, considering future time only, are discussed. The most relevant rules of the approach are presented.

To focus thought, the exposition is exemplified on one of the running examples, serializability. A trace is also, slightly abusing notation, used as a formula characterizing the state after the execution of its actions. It is shown how to establish the following formula:

$$AG([\varepsilon] \rightarrow EF([a_1a_2a_3] \vee [b_1b_2b_3])).$$

To do so, one has to first show how to establish formulas of the form $AG(\varphi \rightarrow EX\psi)$.

Proof Rule for $AG(\varphi \rightarrow EX\psi)$

The semantical meaning of $AG(\varphi \rightarrow EX\psi)$ is: for every run, for every trace τ in that run satisfying φ, there is an action a such that $\tau[a]$ is in that run and satisfies ψ.

To establish this formula, it suffices to provide a set F of actions with the following properties.

For each trace τ satisfying φ,

- for each run which contains τ, there is an action from F extending τ within the run;
- extended traces satisfy ψ.

Such a set is called a *forward intercepting* set.

For the serializability example

- The formula to be established is $AG([\varepsilon] \rightarrow EX([a_1]))$.
- The required set is $\{a_1\}$.

Runs are difficult to handle in a finitary way. Therefore, the first idea is to describe the set of actions without having to consider runs. In order to do so, notice that the observations of a run are equivalent modulo permuting independent actions. The requirement about the set F can thus be expressed as follows.

For each trace τ satisfying φ, for each extension, i.e., action sequence u such that $\tau[u]$ is a trace of the program (thus extending into all runs), one of the following two options should hold.

(OK -Now). The extension contains some action a from F that can be permuted to occur immediately after the original trace, i.e., for all b in u that precede a, $(a, b) \in I$.

(OK -Later). The extension is not yet long enough to contain an action of F but there is some action from F that is concurrently enabled in the extension, i.e., $\tau \models en_a$ and for all b in v, $(a,b) \in I$. (Note that this option may hold for all finite prefixes of an infinite sequence of traces without a ever be taken: in that case the sequence is not an observation and nothing needs to be shown. Also note that because of the consistency requirements a remains enabled throughout v.)

For the serializability example:

- The first option covers the sequences a_1, b_1a_1 and their extensions.
- The second one covers b_1, b_1b_2 and $b_1b_2b_3$.

An extension is called *violating*, if it contains no actions from F that can be permuted back to its beginning and has no concurrently enabled action from F. This requirement can be formulated negatively as: there are no violating extensions.

Now, there are still infinitely many extensions to consider. The second idea is to show that no violating extensions exist by providing a finite graph that generates all potentially harmful candidates.

The nodes of the graph are of form $G_i =< \varphi_i, F_i >$. The intuitive meaning is the following. φ_i is a first order formula that denotes which traces could have been generated when arriving at this node. F_i indicates which actions from F are still enabled at this node.

The edges of the graph carry the transitions that build potentially violating extensions.

The structure of the graph is, intuitively, as follows.

1. The starting node allows all traces τ such that $\tau \models \varphi$ and also enables all actions from F.
2. For each node G_i, every enabled transition $a \notin F_i$ contributes to a potentially violating sequence and therefore is present on the edge to some G_j. To comply with the above intuition, the corresponding φ_j then allows at least all traces that are obtained by extending already generated ones by a. Also, just those actions from F_i that can not be permuted back over a are removed from F_i to yield F_j.
3. To keep the graph finite, it is required that different nodes have different sets of enabled actions.
4. Extensions generated by such a graph are violating ones just if a node G_i is reached where $F_i = \emptyset$. Therefore, in that case $\varphi_i = false$ should hold.
5. At each state satisfying φ every action a in F satisfies the weakest precondition of ψ, $wp_a(\psi)$, i.e., is enabled and, if executed, yields ψ.

The corresponding proof rule is NEXT.

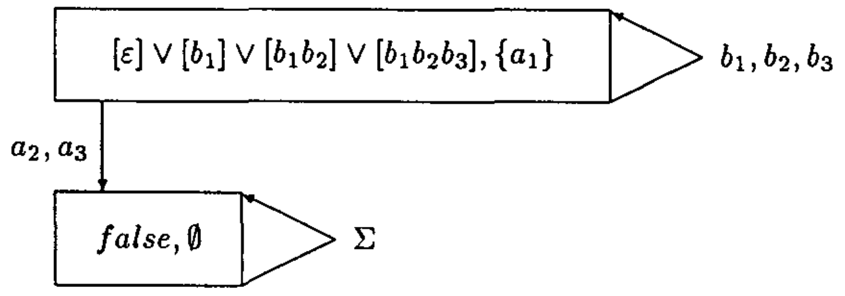

$[\varepsilon] \vee [b_1] \vee [b_1b_2] \vee [b_1b_2b_3], \{a_1\}$
b_1, b_2, b_3
a_2, a_3
$false, \emptyset$
Σ

Figure 24: Graph for the serializability example

N1. $G_0 = (\varphi_0, F_0)$, such that $AG(\varphi \rightarrow \varphi_0)$

N2. If $G_i \xrightarrow{a} G_j$ then $(\varphi_i \wedge en_a) \rightarrow wp_a(\varphi_j)$ and $F_j = F_i - \{b \mid (a,b) \in D\}$

N3. If $G_i \neq G_j$, then $F_i \neq F_j$

N4. If $F_i = \emptyset$, then $\varphi_i = false$

N5. $AG(\varphi \rightarrow wp_a(\psi))$, for each $a \in F$

$AG(\varphi \rightarrow EX\psi)$

For the serializability example, to establish $AG([\varepsilon] \rightarrow EX([a_1]))$, the graph is shown in Figure 24.

A proof rule for $AG(\varphi \rightarrow EF\psi)$

The difference with the NEXT situation is that now ψ should be satisfied in the present in any subsequent state rather than after the execution of a single action.

The idea is to establish in a novel inductive manner that some property ψ holds for every run for some observation in some future state. It turns out that just applying induction forward in time, i.e., providing intermediate assertions $EX\psi_i$ that hold along observations serving as an induction path to the desired state and together achieve ψ fails.

The reason is the implicit universal quantification over runs. Because of this, where runs overlap, intermediate assertions that hold along the observation serving as the induction path to the desired state for one run, may be required to also hold for some second run (for which that observation was never destined to be the induction path). There are situations that this requirement can not be met, i.e., in which the straightforward approach fails. There are cases in which in some state along the induction path of the first run no intermediate assertion can be found at all, because the desired state in the second run can no more be reached. There are situations that this is the case for all induction paths, i.e., situations where the straightforward inductive approach fails.

Serializability again provides an example. To prove is:

$$AG([\varepsilon] \rightarrow EF([a_1a_2a_3] \vee [b_1b_2b_3])).$$

The first intermediate assertion, suitable for the thick-lined run in the Figure 2, would be $AG([\varepsilon] \rightarrow EX[a_1])$ (or, equivalently, $AG([\varepsilon] \rightarrow EX[b_1])$) - choosing $[a_1] \vee [b_1]$ would only worsen the situation.

The next step would then require $AG([a_1] \rightarrow EX[a_1b_1])$. Now the assertion for the subsequent step, $AG([a_1b_1] \rightarrow EF([a_1a_2a_3] \vee [b_1b_2b_3]))$, does not hold anymore.

Already at state $[a_1]$ in fact only $[a_1a_2a_3]$ is reachable anymore, satisfying the requirement for only the thick-lined run. However, at this state the runs still overlap, thus requiring extensions into both. It is evident, that no other choice of inductive paths and corresponding intermediate assertions would escape this situation.

A closer analysis indicates how to overcome the problem.

As runs are directed, there must be some turning state further on in the second run that

1. still can be reached and

2. from which, going backward, the desired state can be reached.

Now if only backward paths are used that subsume the origin where one started, this implies that there is also a path from the origin along which the desired state can be reached by just going forward. That the origin is subsumed is necessary, as otherwise this approach might not be sound; only a forward/backward but no forward path leading to the desired state might exist. The idea is to find an assertion characterizing the turning state and then to show that the desired state can be reached by standard induction applied forward as well as backward.

For the serializability example, from $[a_1]$, and also even from $[a_1b_1]$, $[a_1a_2a_3b_1]$ and $[b_1b_2b_3a_1]$ can both be reached. From these states, there are backwards paths to the desired $[a_1a_2a_3]$ and $[b_1b_2b_3]$. As $[\varepsilon]$ was the origin state, no subsystems need be considered for the backwards paths.

That backward paths subsume the origin is achieved by changing for the backward part to the subsystem rooted in the state where φ held. In [34] this is formalized, here just the notation to indicate this subsystem, P_φ, is borrowed.

The corresponding proof rule is FRBK.

F1 $AG(\varphi \rightarrow EF\hat{\psi})$

$$\frac{\text{F2 } P_\varphi \vdash AG(\hat{\psi} \rightarrow EP\psi)}{AG(\varphi \rightarrow EF\psi)}$$

For the example, by repeated application of the rule NEXT and the use of, essentially, variants of familiar future time temporal logic rules, $F1$ can be established: $AG([\varepsilon] \rightarrow EF([a_1a_2a_3b_1] \vee [b_1b_2b_3a_1]))$. Using similar backward rules, $F2$ is established: $AG([a_1a_2a_3b_1] \vee [b_1b_2b_3a_1] \rightarrow EP([a_1a_2a_3] \vee [b_1b_2b_3]))$. The desired result then follows directly by FRBK.

The need to go to a subsystem for the backward part can also be seen from the example. Consider, for instance, starting from $[a_1]$ rather than $[\varepsilon]$. Again a forward path to $[b_1b_2b_3a_1]$ exists, as well as a backward one to $[b_1b_2b_3]$. However, a path from $[a_1]$ to $[b_1b_2b_3]$ does not exist.

7.5 Proof System to Exploit Abstraction

The first idea is that in certain cases to establish $AG(\varphi \rightarrow AF\psi)$ for a program it is sufficient to prove the weaker property $AG(\varphi \rightarrow EF\psi)$. Namely, for certain combinations of programs and properties. This might especially simplify proofs, when for some convenient observations the desired property is easier to establish than for others.

Unfortunately, the rules for proving EF-properties as discussed in the previous section are still very complicated. The second idea is therefore, to simplify the rules as well. Rules are simplified in the sense that they remain sound for all programs and properties, but generally not complete. Of course again a, maybe different, subclass of programs and properties could be provided for which such rules are complete.

The question of simplifying the rules is addressed first. Secondly, a class of programs and properties is defined for which OF properties imply OF-ones. It turns out that for this class the simplified rules are in fact complete.

The inevitability property of the second example program is used to illustrate the approach.

A simplified rule for $AG(\varphi \rightarrow EF\psi)$

To limit the number of actions to be considered when iteratively assessing representative sequences, for each state τ a subset H_τ of helpful actions is defined.

The idea is to find a set of actions that are helpful in that in any state satisfying φ, each of them yields ψ. The set should also be such that in each run there is an observation where such an action is executed at the right moment, i.e., in a state where φ holds.

The former requirement will be seen to be incorporated in the premises of a proof rule. The latter requirement is achieved by the following definition, provided that an action from the set is continuously enabled.

Definition 7.5 *A* faithful decomposition *of a program* in a state τ *is a subset of actions* $\mathcal{H}_\tau \subseteq \Sigma$ *such that each action in its complement,* $\overline{\mathcal{H}_\tau}$*, is either independent of each action in* $\mathcal{H}_\tau$ *or disabled in* τ *and its successors as long as no action from* $\mathcal{H}_\tau$ *is executed.*

Let faithful$(\varphi, \mathcal{H})$ *denote that* $\mathcal{H}$ *is a faithful decomposition in each state satisfying* φ.

The idea is then to establish $AG(\varphi \rightarrow EF\psi)$ as follows. Find a set $\mathcal{H}$ with the following properties.

1. To ensure availability of helpful actions, $\mathcal{H}$ should be faithful for φ.

2. To ensure enabledness of a helpful action, there should be an action a in $\mathcal{H}$ that is enabled if φ holds.

   Furthermore, the following should hold about this action.

3. Actions from the faithful set that are independent from a need not be helpful themselves, as they leave a enabled - they should leave φ true though, as otherwise a might not yield ψ anymore. Of course, if "perchance" they yield the desired ψ, this is allowed too.

4. Actions from $\mathcal{H}$ that are not independent from a, for instance a itself, should yield ψ. This is required, because such actions might disable actions from $\mathcal{H}$.

This leads to the following rule.

EVENTUAL

E1. $faithful(\varphi, \mathcal{H})$,

For some $a \in \mathcal{H}$,

E2. $AG(\varphi \rightarrow en_a)$;

E3. for each $b \in \mathcal{H}$ such that $(b,a) \in I, AG((\varphi \wedge en_b) \rightarrow wp_b(\varphi \vee \psi))$;

E4. for each $b \in \mathcal{H}$ such that $(b,a) \notin I, AG((\varphi \wedge en_b) \rightarrow wp_b(\psi))$

$AG(\varphi \rightarrow EF\psi)$

Using this rule to prove $AG(\neg(a \in \Sigma(\tau) \vee b \in \Sigma(\tau)) \rightarrow EF(a \in \Sigma(\tau) \vee b \in \Sigma(\tau)))$ for the inevitability example, $faithful(\neg(a \in \Sigma(\tau) \vee b \in \Sigma(\tau)), \{a,b,c,d\})$ is chosen. The premises of the EVENTUAL rule are then fulfilled:

E1. {a,b,c,d} is trivially faithful, as $\overline{\{a,b,c,d\}} = \emptyset$

Choose a as the special action in $\{a,b,c,d\}$,

E2. indeed a is enabled if it and b have not occurred yet;

E3. the actions from $\{a,b,c,d\}$ that are independent of a, i.e., c and d, preserve $\neg(a \in \Sigma(\tau) \vee b \in \Sigma(\tau))$;

E4. the only action from $\{a,b,c,d\}$ that is not independent of a, b, yields $(a \in \Sigma(\tau) \vee b \in \Sigma(\tau))$

$AG(\neg(a \in \Sigma(\tau) \vee b \in \Sigma(\tau)) \rightarrow EF(a \in \Sigma(\tau) \vee b \in \Sigma(\tau)))$

A condition under which replacement of E by A is allowed

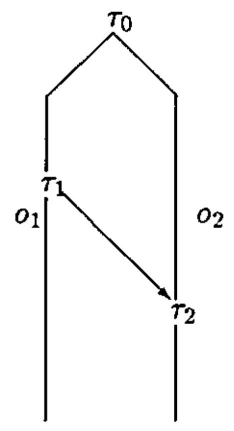
To allow E in $EF\psi$ to be replaced by A, the following should hold for the combination of program P and properties φ and ψ. For each run, there cannot be observations o_1 and o_2 starting from a state τ_0 where φ holds such that ψ holds eventually in some state τ_1 on o_1, but $\neg\psi$ continually holds on o_2. From this, a sufficient, more manageable, condition is now argued.

Assume such an offending pair of observations exists. Because o_1 and o_2 are observations of the same run, there is a state τ_2 on o_2 that is reachable from τ_1. The resulting path p_1 from τ_0 to τ_2, shown in Figure 25, and the path p_2 from τ_0 to τ_2 on o_2 form a large diamond, and hence are equivalent. Therefore, p_2 can be transformed into p_1 by repeatedly exchanging the order between adjacent independent actions - "flipping". Always flipping the lowest of such pairs, one of these flippings will cause, on a prefix of p_2, the first occurrence of a state where ψ holds. Hence there exists a diamond on o_2, defined as follows.

Definition 7.6 *An offending diamond for ψ is a state in which $\neg\psi$ holds, with two immediate successors τ' and τ'', generated by two independent actions a and b. The state τ' satisfies ψ and the state τ'' satisfies $\neg\psi$. Furthermore, the b-successor of τ', which is also the a-successor of τ'', satisfies $\neg\psi$.*

This leads to the following theorem (see Figure 25).

Theorem 7.7 *The non-existence of offending diamonds is sufficient to ensure that $AG(\varphi \rightarrow EF\psi)$ implies $AG(\varphi \rightarrow AF\psi)$.*



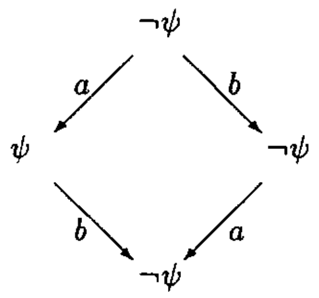


Figure 25: An offending pair and an offending diamond

For the inevitability example: There exists no offending diamond, as once the program establishes $a \in \Sigma(\tau) \vee b \in \Sigma(\tau)$, this can not be made undone.

Therefore, $AG(\neg(a \in \Sigma(\tau) \vee b \in \Sigma(\tau)) \rightarrow AF(a \in \Sigma(\tau) \vee b \in \Sigma(\tau)))$ has now been shown.

For the class of programs and properties defined by the requirement that no offending diamond exists, the EVENTUAL rule (together with the other rules) is complete in the following sense. If $\mathcal{H} = \Sigma$ is chosen, the rule reduces to the LTL rule to prove $AG(\varphi \rightarrow AF\psi)$. For the restricted class of programs and properties, this is exactly what the EVENTUAL rule achieves.

8 Axiomatization of a Subset of Propositional ISTL$_P$

In this section, a complete infinitary proof system for a subset of propositional ISTL with past operators is presented.

In Section 6, it has been shown how to reduce the validity problem for CTL$_P$ to the recurring tiling problem. This proof also goes through for ISTL$_P$, because the grid encoded has the forward-diamond property and ISTL with past operators is at least as expressive as CTL$_P$ over such models. The complexity of the recurring tiling problem, and thus also of the validity problem, is known to be Π^1_1-hard. Hence, a complete finitary axiomatization of ISTL$_P$ does not exist. This motivates the presence of infinitary rules.

In what follows, only trace systems with infinite runs are dealt with. This inessential restriction allows to consider only infinite models for ISTL$_P$ and admits of a simpler axiomatization.

The part of the proof system concerned with characterizing the logical operators is standard, as for CTL with past and labelled operators.

The part of the axiom system concerned with characterizing models is obtained as follows. A frame for ISTL$_P$ corresponds to a run of a trace system. Properties of ISTL$_P$ frames given in Lemma 2.11, where the forward-I_F-diamond property $C4$ is strengthen to the forward-diamond property b) of Definition 2.12, are axiomatized. The only property which cannot be expressed in the logic and therefore axiomatized is $C1$.

The present exposition is based on the paper by Penczek [38], where the complete proof system has been given for a subset of ISTL$_P$ extended with the past formulas of form $EH\varphi$ and $E\varphi S\psi$.

8.1 Proof System for ISTL'$_P$

As an intermediate step, a proof system is given for the logic ISTL'$_P$ which is like ISTL$_P$ except for the fact that the forward path quantifiers range over maximal paths rather than over observations, i.e., in the semantic rule $S3$ of ISTL the word "observation" is replaced by "maximal path".

The current literature is followed in that a proof system is given for the floating version of the logic, i.e., the logic with validity (denoted $\models_f$) defined over all models and all states. Both (floating and anchored) versions are of the same expressive power, and a proof system for either one can be used as a proof system for the other, using the following translation rules.

1. $\models \psi$ iff $\models_f AY false \rightarrow \psi$,
2. $\models_f \psi$ iff $\models AG\psi$.

1) says that ψ holds in all models at the beginning states iff $AY false \rightarrow \psi$ holds in all models at all states. Note that the formula trivializes at all but the beginning state. 2) expresses that ψ holds in all models at all states iff $AG\psi$ holds in all models at the beginning states.

Denote by:

- $EX^i(\varphi) \overset{def}{=} \varphi \wedge EX(\varphi \wedge EX(\varphi \wedge ...EX(\varphi)...)$
  (the operator EX occurs i times, for $i \geq 0$),
- $EX^0(\varphi, \psi) \overset{def}{=} \psi$, $EX^i(\varphi, \psi) \overset{def}{=} \varphi \wedge EX(\varphi \wedge EX(\varphi \wedge ...EX(\psi)...)$
  (the operator EX occurs i times, for $i \geq 1$),
- $EX_\epsilon\varphi \overset{def}{=} \varphi$, $EX_u\varphi \overset{def}{=} EX_{a_1}EX_{a_2}...EX_{a_n}\varphi$, for $u = a_1a_2...a_n$.
- $I(a,b) \overset{def}{=} EPEF(EY_a true \wedge EY_b true)$, for $a \neq b$, $a, b \in \Sigma$.

$I(a,b)$ expresses that the actions a and b are executed independently in the model.

Axioms

A1. all formulas of the form of tautologies of the classical prop. calculus

A2. $EG\varphi \equiv \varphi \wedge EX(EG\varphi)$ (fixed-point characterization of EG)

A3. $E(\varphi U \psi) \equiv \psi \vee (\varphi \wedge EX(E(\varphi U \psi)))$ (fixed-point characterization of EU)

A4. $EP\psi \equiv \psi \vee EYEP\psi$ (fixed-point characterization of EP)

A5. $\varphi \rightarrow AX_aEY_a\varphi$ (relating past and future)

A6. $\varphi \rightarrow AY_aEX_a\varphi$ (relating past and future)

A7. $EXtrue$ (infiniteness)

A8. $EP(AYfalse)$ (beginning)

A9. $EX_a(\varphi \wedge \psi) \equiv EX_a(\varphi) \wedge EX_a(\psi)$ (forward determinism)

A10. $EX_aAX_b\varphi \rightarrow AX_bEX_a\varphi$, for $a \neq b$ (forward-diamond property)

A11. $(I(a,b) \wedge EX_aEX_b\varphi) \rightarrow EX_bEX_a\varphi$ (concurrency closure property)

A12. $EY_a(\varphi \wedge \psi) \equiv EY_a(\varphi) \wedge EY_a(\psi)$ (no auto-concurrency)

A13. $EY_aAY_b\varphi \rightarrow AY_bEY_a\varphi$, for $a \neq b$ (backward-diamond property)

Proof Rules

MP. $\varphi, \varphi \rightarrow \psi \vdash \psi$ (modus ponens)

R1. $\varphi \rightarrow \psi \vdash EX_a\varphi \rightarrow EX_a\psi$

R2. $\varphi \rightarrow \psi \vdash EY_a\varphi \rightarrow EY_a\psi$

R3. $\{\phi \rightarrow EX_uEX^i(\varphi)\}_{i \in \omega} \vdash \phi \rightarrow EX_uEG\varphi$, for $u \in \Sigma^*$

R4. $\{EX_uEX^i(\varphi, \psi) \rightarrow \phi\}_{i \in \omega} \vdash EX_uE(\varphi U \psi) \rightarrow \phi$, for $u \in \Sigma^*$

R5. $AYfalse \rightarrow AG\varphi \vdash \varphi$.

MP is the standard modus ponens rule. $R1$ and $R2$ are rules expressing deductive closures. $R3$ and $R4$ are infinitary rules characterizing EG and EU. $R5$ characterizes the beginning state.

The "only if" part of the following lemma is used to prove that $R3$ and $R4$ preserve validity.

Lemma 8.1 *For every model M and each state w,*

(a) $M, w \models EG\varphi$ iff $M, w \models EX^i(\varphi)$, for each $i \in \omega$,

(b) $M, w \models E(\varphi U \psi)$ iff $M, w \models EX^i(\varphi, \psi)$, for some $i \in \omega$.

Proof: a) ($\Rightarrow$) follows directly from the definition of the temporal operator EG.

Since Σ is finite, each state in M has only finitely many successors. Therefore, a) ($\Leftarrow$) follows from König's lemma.

b) similarly follows directly from the definitions of the operators EU and $EX^i(.,.)$. □

Note that the use of maximal paths rather than observations is crucial here; the right hand sides of the two clauses guarantee the existence of a maximal path, but not necessarily of an observation.

Theorem 8.2 *The proof system is sound (i.e., $\vdash \varphi$ implies $\models_f \varphi$).*

Proof: It is easy to check that the axioms are valid and the proof rules preserve validity. For the rules $R3$ and $R4$, this follows from Lemma 8.1 and the forward determinism of frames. □

Completeness of the Proof System for ISTL'$_P$

The construction in this section builds a model for a consistent formula. An outline of the construction is as follows:

- The Lindenbaum-Tarski algebra for the logic is built.
- The infinite operations, say Q, corresponding to the infinitary rules are defined in the algebra.
- Q-filters in the algebra are defined. These are then used as worlds of the model.
- The model is built. The axioms are used to prove that the model is isomorphic with a run.

Let $\sim \subseteq Form \times Form$ be the following relation: $\varphi \sim \psi$ iff $\vdash \varphi \equiv \psi$.

Note that $\sim$ is a congruence with respect to all the logical and modal operators. Let $Form/\sim$ denote the set of all equivalence classes of $\sim$. Elements of $Form/\sim$ are denoted as follows $[\varphi], [\psi], \ldots$

Definition 8.3 *The* Lindenbaum-Tarski algebra *for ISTL'$_P$ is a 6-tuple $LTA = (Form/\sim, \cup, \cap, -, [true], [false])$, where*

- $[\varphi] \cup [\psi] = [\varphi \vee \psi]$,
- $[\varphi] \cap [\psi] = [\varphi \wedge \psi]$,
- $-[\varphi] = [\neg\varphi]$.

Theorem 8.4 *The Lindenbaum-Tarski algebra LTA satisfies the following conditions:*

1. LTA is a non-degenerate Boolean algebra,

2. $\vdash \varphi$ *iff* $[\varphi] = [true]$,

3. $\nvdash \varphi$ *iff* $[\neg\varphi] \neq [false]$.

The proof of the above theorem is standard and can be found in [40] (p. 257). Let $\leq$ be a partial ordering in $Form/\sim \times Form/\sim$, defined as follows:

$$[\varphi] \leq [\psi] \ iff \ \vdash \varphi \to \psi.$$

Lemma 8.5 *In the algebra LTA the following conditions hold:*

(a) $[EX_u EG\varphi] = inf_{i\in\omega}\{[EX_u EX^i\varphi]\}$, *for* $u \in \Sigma^*$,

(b) $[EX_u E(\varphi U\psi)] = sup_{i\in\omega}\{[EX_u EX^i(\varphi,\psi)]\}$, *for* $u \in \Sigma^*$.

Proof: Follows from axioms $A2$, $A3$, and rules $R3$ and $R4$. □

Let Q denote the following infinite operations in the algebra LTA: $inf_{i\in\omega}\{EX_u EX^i(.)\}$, and $sup_{i\in\omega}\{EX_u EX^i(.,.)\}$, for $u \in \Sigma^*$.

Definition 8.6 *A Q-*filter *in LTA is a maximal proper filter A which satisfies the following conditions:*

- *if* $[EX_u E(\varphi U\psi)] \in A$, *then there is* $i \in \omega$ *such that* $[EX_u EX^i(\varphi,\psi)] \in A$,
- *if* $[EX_u EG\varphi] \notin A$, *then there is* $i \in \omega$ *such that* $[EX_u EX^i\varphi] \notin A$.

Notice that the number of infinite operations is enumerable. The following standard result is assumed (see [40]).

Lemma 8.7 *If a set Q of infinite operations in a Boolean algebra is at most enumerable, then every non-zero element of the Boolean algebra belongs to a Q-filter.*

Completeness can now be argued.

Theorem 8.8 *The proof system is complete (i.e.,* $\models_f \phi$ *implies* $\vdash \phi$*).*

Proof. (sketch)
It is shown that if $\nvdash \phi$, then there is a model for $\neg\phi$, which by contraposition implies that the theorem holds. Let UF be the family of all ultrafilters in the algebra LTA and $QF \subseteq UF$ be the family of all Q-filters in LTA. Define the following relation in $UF \times \Sigma \times UF$:

- $D \xrightarrow{a}_F D'$ iff $\{[EX_a\varphi] \mid [\varphi] \in D'\} \subseteq D$.

The following lemma is needed in order to build a required model for $\neg\phi$.

Lemma 8.9 *If D is a Q-filter and* $[EX_a\varphi] \in D$, *then there is the Q-filter D' s.t.* $D \xrightarrow{a}_F D'$ *and* $[\varphi] \in D'$.

Proof: Using the standard construction method (cf. [40]), an ultrafilter D' can be built s.t. $D \xrightarrow{a}_F D'$ and $[\varphi] \in D'$. To show that D' is a Q-filter, assume that $[EX_uE(\psi U\phi)] \in D'$. Then, by definition of $\xrightarrow{a}_F$, $[EX_aEX_uE(\psi U\phi)] \in D$. Since D is a Q-filter, there is $i \in \omega$ such that $[EX_aEX_uEX^i(\psi,\phi)] \in D$. Therefore, by $A9$, $[EX_uEX^i(\psi,\phi)] \in D'$. To complete the proof that D' is a Q-filter, assume that $[EX_uEG\psi] \notin D'$. Then, by definition of $\xrightarrow{a}_F$, $[EX_aEX_uEG\psi] \notin D$. Since D is a Q-filter, there is $i \in \omega$ such that $[EX_aEX_uEX^i\psi] \notin D$. Therefore, $[EX_uEX^i\psi] \notin D'$, which proves that D' is a Q-filter.

It follows from $A9$ that D' is the only ultrafilter s.t. $D \xrightarrow{a}_F D'$. □

Then, if $\nvdash \phi$, then, by $R5$, $\nvdash AYfalse \to AG\phi$. Then, by Theorem 8.4, $[AYfalse \wedge \neg AG\phi] \neq [false]$. Therefore, by Lemma 8.7, there is a Q-filter $C \in QF$ such that $[AYfalse] \in C$ and $[EF(\neg\phi)] \in C$. Next, the structure $F = (W, \Sigma, \to, C)$ and the structure $M = (F, V)$ is defined as follows:

- $W = \{D \in QF \mid \exists D_1, ..., D_n \in QF,\ D_1 \to_F ... \to_F D_n,\ D_1 = C,\ D_n = D\}$,
- $\to\ = \ \to_F \cap (W \times \Sigma \times W)$,
- $q \in V(D)$ iff $[q] \in D$, where $q \in PV$ and $D \in W$.

Now, the proof can be completed.

Lemma 8.10 *The following conditions hold:*

a) F is a frame for $ISTL_P$,

b) For each $\varphi \in Form$, $D \in W$: $M, D \models \varphi$ iff $[\varphi] \in D$.

Proof: a) All the conditions stated in Lemma 2.11, where the forward-I_F-diamond property $C4$ is strengthen to the forward-diamond property b) of Definition 2.12, hold for F.

- $C1$ holds by the definition of W; F is infinite by $A7$,
- For $2 \leq i \leq 7$, Ci holds by the axiom Ak, where $k = i + 6$.

b) The proof is by induction on the complexity of a formula φ using a well-founded relation in the set of formulas. □

Therefore, M is a model containing a state satisfying $\neg\phi$, which completes the proof of completeness.

8.2 Proof System for a Subset of ISTL$_P$

In the original semantics of ISTL$_P$, the quantifiers E and A range over observations rather than over forward paths. It is now shown that a small modification of the proof system for ISTL'$_P$ leads to a complete proof system for the subset of ISTL$_P$ without formulas of the form $EG\varphi$, but with the formulas of the form $E(\varphi U\phi)$ and $A(\varphi U\phi)$.

Let A_o and E_o denote the quantifiers ranging over observations. $E_o(\varphi U \psi)$ and $A_o(\varphi U \psi)$ can be expressed in the defined language and therefore derived using the proof system.

Firstly, observe that $M, w \models E_o(\varphi U \psi)$ iff $M, w \models E(\varphi U \psi)$. Unfortunately, the analogous property does not hold for $A_o(\varphi U \psi)$. This problem can be solved as follows. Select a proposition $\sigma \in PV$ and add the following three new axioms characterizing its values in the models:

$A\sigma 1.$ $AY false \rightarrow \sigma$,

$A\sigma 2.$ $\sigma \rightarrow EX_! \sigma \wedge (AY false \vee EY_! \sigma)$,

where $EX_! \sigma \overset{def}{=} \bigoplus_{a \in \Sigma} EX_a \sigma$, $EY_! \sigma \overset{def}{=} \bigoplus_{a \in \Sigma} EY_a \sigma$.

$A\sigma 3.$ $EF\sigma$.

The axioms $A\sigma$1-2 express that σ holds in exactly one path in the model. $A\sigma 3$ ensures that the path at which σ holds is an observation, namely, by stating that it is cofinal with the run.

Now, the formulas of the form $A_o(\varphi U \psi)$ are shown to be expressible in the defined language:

Theorem 8.11 *The following equivalence holds:*

$$\models_f A_o(\varphi U \psi) \ \ iff \ \ \models_f \sigma \rightarrow E((\varphi \wedge \sigma) U (\psi \wedge \sigma)),$$

where ψ, φ and ψ do not contain σ.

The proof of ($\Leftarrow$) follows from the fact that the formula $(\varphi U \psi)$ must hold in each model at the observation marked σ. Since for each run and each its observation there is a model in which this observation is marked σ, $(\varphi U \phi)$ holds in all models at all observations.

In order to express formulas containing several subformulas of the form $A_o(\varphi_i U \phi_i)$, for each of them a special proposition σ_i has to be selected and three new axioms for σ_i, are added, as above.

Therefore, the extended proof system contains a complete axiomatization of ISTL$_P$ without formulas of the form $E_o G \varphi$.

Moreover, it follows from the completeness theorem that the set of theorems of the axiomatized fragment of ISTL is at most Π^1_1. Therefore, the validity problem for this subset is Π^1_1-complete.

The given proof system for ISTL$_P$ can be adapted to a proof system for CTL$_P$ [38] by strengthening $A10$ to the axiom

$(I(a, b) \wedge EX_a AX_b \varphi) \rightarrow AX_b EX_a \varphi$, for $a \neq b$ (I-diamond property).

9 Conclusions

Except for the introductory material, the table below follows from the organization of the material in this chapter. The possible entries generated by the tabular presentation provide some guidance to assess the state-of-the-art. In the following table ?? indicates current absence of a result; - indicates that a slot can not be filled in a meaningful way.

	model checking				proof rules	
logic	standard	efficient	decidable	axiomatization	standard	efficient
LTL	YES	YES	YES	YES	YES	YES
CTL	YES	YES	YES	YES	YES	??
CTL*	YES	YES	YES	??	??	??
ISTL	??	-	??	??	??	-
ISTL*	??	-	NO	??	??	-
$ISTL_P$	??	-	NO	YES	YES	-
$ISTL*_P$	??	-	NO	??	??	-
CTL_P	YES	-	NO	YES	??	-
$CTL*_P$	??	-	NO	??	??	-

The linear temporal logic LTL and its extension CTL* that enables to identify forward branching points are conventional temporal logics in the sense that they do not address partial order properties. The restriction CTL of CTL* is included because in that case the complexity of standard model checking is linear. The information about the existence of standard model checking procedures, axiomatizations, decision procedures and program proof rules for these logics is included in the table not so much because of direct relevance to partial order considerations, but to provide a somewhat wider context. Most of the results are discussed in [6].

As shown in Sections 4 and 5, partial order techniques can make model checking more efficient for these logics without next step operators. Note that efficient model checking is based on the idea that logics can only distinguish up to permutations of independent actions, plus that in case of CTL also branching points are relevant. Formulas can therefore be checked over any acceptor that respects the corresponding equivalence. The independence of actions is thus only used for reducing the acceptors; the logics considered are not partial order logics themselves. Because of the very fact that the logics ISTL and CTL_P and their extensions do enable to distinguish between permutations, efficient model checking is not an option, as indicated in the table. Methods based on ample sets as well as on sleep sets are presented for LTL. The adaptation of the ample set approach for CTL (and, as only reducing acceptors is involved, also for CTL*) is quite recent; the sleep set direction has not yet been explored.

The same remark about enhancing efficiency as made in the case of model checking applies in principle to program proof rules: It is the efficiency of proving LTL formulas about programs that is enhanced, again the approach does not apply for ISTL* and CTL_P. There is a difference though, in that model checking is a semantic activity but proving formulas occurs at the syntactic level. Therefore, independence of actions cannot be hidden in the model but needs to be present in the proof system. So in fact proving an LTL formula efficiently amounts to proving a formula from a subset of ISTL* formulas. The table indicates what is in fact achieved: efficient proving of LTL formulas about programs.

Program proof systems for branching time logics were slow to develop. The proof system Fix and Grumberg provide for fair CTL [9] is the first result in this area. Whether or not simplifying proof rules for CTL and extensions to CTL* are feasible remains yet to be seen.

ISTL* extends the expressive power of LTL by enabling to explicitly mention representatives of runs, i.e., exploit independence of actions. $\mathrm{ISTL}^*{}_P$ [34] is an extension of ISTL* with CTL*-like nested past operators. As discussed in Section 7, on the one hand syntactically restricted and on the other hand extended, namely with past operators, version ISTL_P, admits of an axiomatization and program proof system. In the absence of the past operators, i.e., for ISTL, no results are available. The program proof rules and axiomatization for ISTL_P is still incomplete in that it does not cover formulas of form $EG\varphi$. The relevance of such formulas for expressing properties of programs is still a matter of debate, though. CTL_P extends the expressive power of CTL by enabling to identify backward branching. This enables to express independence of actions; together with the capability to identify forward branching already present in CTL, properties of runs can be described, be it in a less direct manner than in the case of ISTL*. Note that in the case of both ISTL_P and CTL_P the axiomatizations are infinitary.

The results in Section 6 about CTL_P are that this logic, exploiting the expressiveness of the trace approach, is undecidable but that NP-hard model checking applies; a one-exponential algorithm is presented for a restricted version without nested past operators.

Some other options are the following. LTL extended with past operators is not discussed, as the interpretation over sequences causes the expressiveness to remain the same as for LTL. QISTL [17] and CCTL [35] can be viewed as ISTL-like extensions of branching rather than linear time logics. As both logics are quite involved and their practical relevance is yet to be assessed more fully, these are not included in the table. A perhaps somewhat less obvious but interesting new direction is proposed by Thiagarajan in [44]. The main idea there is to interpret a multi-agent linear temporal logic over independence graphs of infinite traces. This approach is motivated by the aim to connect decidability and model checking problem for the logic to testing for non-emptiness of asynchronous Büchi automata.

References

[1] M. Bednarczyk, *Categories of Asynchronous Transition Systems*, PhD thesis, University of Sussex, 1987, Available as Report 1/88, School of Cognitive and Computing Sciences, University of Sussex.

[2] M. C. Browne, E. M. Clarke, and O. Grümberg, Characterizing finite Kripke structures in propositional temporal logic, *Theoretical Computer Science*, 59:115–131, 1988.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla, Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach, *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[4] M. Droste, Concurrency, automata and domains, In M. S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90), Warwick (England) 1990*, number 443 in Lecture Notes in Computer Science, pages 195–208, Berlin-Heidelberg-New York, 1990, Springer.

[5] V. Diekert, P. Gastin, and A. Petit, Rational and recognizable complex trace languages, *Information and Computation*, to appear.

[6] E. A. Emerson, Temporal and modal logic, In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072, Elsevier Science Publisher B. V., 1990.

[7] E. A. Emerson and J. Y. Halpern, Decision procedures and expressiveness in the temporal logic of branching time, *Journal of Computer and System Sciences*, 30:1–24, 1985.

[8] E. A. Emerson and J. Srinivasan, Branching time temporal logic, In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in Lecture Notes in Computer Science, pages 123–172, Berlin-Heidelberg-New York, 1988, Springer.

[9] L. Fix and O. Grumberg, Verification of temporal properties, Technical Report TR 93-1368, CS Cornell Univ., Ithaca NY, Aug. 1993.

[10] P. Gastin, Infinite traces, In I. Guessarian, editor, *Proceedings of the Spring School of Theoretical Computer Science on Semantics of Systems of Concurrent Processes*, number 469 in Lecture Notes in Computer Science, pages 277–308, Berlin-Heidelberg-New York, 1990, Springer.

[11] R. Gerth, R. Kuiper, D. Peled, and W. Penczek, A partial order approach to branching time logic model checking, In *Proceedings of the Israeli Conference on Theoretical Computer Science*, 1995.

[12] P. Godefroid, Using partial orders to improve automatic verification methods, In E. M. Clarke, editor, *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, pages 176–185, Berlin-Heidelberg-New York, 1991, Springer.

[13] P. Godefroid and P. Wolper, A partial approach to model checking, In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[14] U. Goltz, R. Kuiper, and W. Penczek, Propositional temporal logics and equivalences, In W. R. Claveland, editor, *Proceedings of the Third International Conference on Concurrency Theory CONCUR'92*, number 630 in Lecture Notes in Computer Science, pages 222–235, Berlin-Heidelberg-New York, 1992, Springer.

[15] D. Harel, Recurring dominoes: Making the highly undecidable highly understandable, *Annals of Discrete Mathematics*, 24:51–72, 1985.

[16] S. Katz and D. Peled, Defining conditional independence using collapses, In M. Z. Kwiatkowska, M. W. Shields, and R. M. Thomas, editors, *Semantics for Concurrency*, Workshops in Computing, pages 262–280, Berlin-Heidelberg-New York, 1990, Springer.

[17] S. Katz and D. Peled, Interleaving set temporal logic, *Theoretical Computer Science*, 75(3):21–43, 1991.

[18] S. Katz and D. Peled, Verification of distributed programs using representative interleaving sequences, *Distributed Computing*, 6:107–120, 1992.

[19] M. Kwiatkowska, *Fairness for non-interleaving concurrency*, PhD Thesis, University of Leicester (UK), 1989.

[20] O. Lichtenstein and A. Pnueli, Checking that finite state concurrent programs satisfy their linear specification, In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[21] K. Lodaya, R. Parikh, R. Ramanujam, and P. S. Thiagarajan, A logical study of distributed transition systems, *Information and Computation*, 1994, to appear.

[22] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Specification*, Springer, 1991.

[23] A. Mazurkiewicz, Concurrent program schemes and their interpretations, DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[24] A. Mazurkiewicz, Trace theory, In W. Brauer et al., editors, *Petri Nets, Applications and Relationship to other Models of Concurrency*, number 255 in Lecture Notes in Computer Science, pages 279–324, Berlin-Heidelberg-New York, 1987, Springer.

[25] A. Mazurkiewicz, E. Ochmański, and W. Penczek, Concurrent systems and inevitability, *Theoretical Computer Science*, 64:281–304, 1989.

[26] M. Nielsen, G. Plotkin, and G. Winskel, Petri nets, event structures and domains, part 1, *Theoretical Computer Science*, 13:85–108, 1981.

[27] E. Ochmański, Regular behaviour of concurrent systems, *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 27:56–67, Oct 1985.

[28] E. Ochmański, Semi-Commutation and Petri Nets, In V. Diekert, editor, *Proceedings of the ASMICS workshop Free Partially Commutative Monoids, Kochel am See 1989*, Report TUM-I9002, Technical University of Munich, pages 151–166, 1990.

[29] E. Ochmański, Modelling concurrency with semi-commutations, In I. M. Havel and V. Koubek, editors, *Proceedings of the 17th Symposium on Mathematical Foundations of Computer Science (MFCS'92), Prague, (Czechoslovakia), 1992*, number 629 in Lecture Notes in Computer Science, pages 412–420, Berlin-Heidelberg-New York, 1992, Springer.

[30] D. Peled, Interleaving set temporal logic, Master thesis, Technion, Israel, 1987.

[31] D. Peled, All from one, one from all: on model checking using representatives, In *Proceedings of the 5th International Conference on Computer Aided Verification, Greece*, number 697 in Lecture Notes in Computer Science, pages 409–423, Berlin-Heidelberg-New York, 1993, Springer.

[32] D. Peled, Combining partial order reductions with on-the-fly model checking, In *Proceedings of 6th International Conference on Computer Aided Verification, Stanford, California*, number 818 in Lecture Notes in Computer Science, pages 377–390, Berlin-Heidelberg-New York, June 1994, Springer.

[33] D. Peled, S. Katz, and A. Pnueli, Specifying and proving serializability in temporal logic, In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS '91)*, 1991.

[34] D. Peled and A. Pnueli, Proving partial order properties, *Theoretical Computer Science*, 126:143–182, 1994, A preliminary version appeared in the proceedings of ICALP'90, Lecture Notes in Computer Science 443, Springer.

[35] W. Penczek, A concurrent branching time temporal logic, In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *Proceedings of the 3rd Workshop on Computer Science Logic*, number 440 in Lecture Notes in Computer Science, pages 337–354, Berlin-Heidelberg-New York, 1990, Springer.

[36] W. Penczek, On temporal logics for trace systems, In V. Diekert and W. Ebinger, editors, *Proceedings ASMICS Workshop Infinite Traces, Tübingen*, Bericht 4/92, pages 158–204, Universität Stuttgart, Fakultät Informatik, 1992.

[37] W. Penczek. On undecidability of temporal logics on trace systems. *Information Processing Letters*, 43:147–153, 1992.

[38] W. Penczek, Axiomatizations of temporal logics on trace systems, In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science (STACS'93), Würzburg 1993*, number 665 in Lecture Notes in Computer Science, pages 452–462, Berlin-Heidelberg-New York, 1993, Springer, the full version submitted to Fundamenta Informaticae.

[39] W. Penczek, Temporal logics for trace systems: On automated verification, *International Journal of Foundations of Computer Science*, 4:31–67, 1993.

[40] H. Rasiowa and R. Sikorski, *The Mathematics of Metamathematics*, PWN, Warszawa, 1970.

[41] W. Reisig, *Petri Nets (an Introduction)*, Number 4 in EATCS Monographs on Theoretical Computer Science, Springer, Berlin-Heidelberg-New York, 1985.

[42] G. Rozenberg, Behaviour of elementary net systems, In W. Brauer, editor, *Petri nets: central models and their properties; advances in Petri nets; proceedings of an advanced course, Bad Honnef, 8.-19. Sept. 1986, Vol. 1*, number 254 in Lecture Notes in Computer Science, pages 60–94, Berlin-Heidelberg-New York, 1986, Springer.

[43] E. W. Stark, Concurrent transition systems, *Theoretical Computer Science*, 64:221–269, 1989.

[44] P. S. Thiagarajan, A trace based extension of linear time temporal logic, In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS'94)*, Lecture Notes in Computer Science, pages 438–447, 1994.

[45] A. Valmari, Stubborn sets for reduced state space generation, In *Proceedings of 10th International Conference on Application and Theory of Petri Nets*, volume 2, pages 1–22, 1989.

[46] A. Valmari, A stubborn attack on state explosion, *Formal Methods in System Design*, 1:285–313, 1992.

[47] M. Y. Vardi and P. Wolper, An automata-theoretic approach to automatic program verification, In D. Kozen, editor, *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 322–331, 1986.

[48] G. Winskel, An introduction to event structures, In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in Lecture Notes in Computer Science, pages 123–172, Berlin-Heidelberg-New York, 1988, Springer.

[49] G. Winskel and M. Nielsen. Models for concurrency. To appear in Handbook of Logic in Computer Science, Oxford University Press.

[50] P. Wolper, On the relation of programs and computations to models of temporal logic, In *Proceedings of the Colloquium on Temporal Logic in Specification, Altrincham, UK, 1987*, number 398 in Lecture Notes in Computer Science, pages 75–123, Berlin-Heidelberg-New York, 1989, Springer.

[51] P. Wolper and P. Godefroid, Partial-order methods for temporal verification, In E. Best, editor, *Proceedings of the Third International Conference on Concurrency Theory CONCUR'93*, number 715 in Lecture Notes in Computer Science, pages 233–246, Berlin-Heidelberg-New York, 1993, Springer.

94/39	A. Blokhuis, T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks, D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet, J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse, M. Bijsterveld	Category Theory as Coherently Constructive Lattice M . Theory: An Illustration, p. 35.
94/44	E. Brinksma, J. Davies, R. Gerth, S. Graf, W. Janssen, B. Jonsson, S. Katz, G. Lowe, M. Poel, A. Pnueli, C. Rump, J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratie-ve Logistiek", p. 43.
94/46	R. Bloo, F. Kamareddine, R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo, F. Kamareddine, R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten, J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks, D. Kratsch, H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.

94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.
94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.

93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Komatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.

93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for $F\omega$, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.