LOGIQUE TEMPORELLE POUR LA SPECIFICATION ET LA VERIFICATION DES SYSTEMES REACTIFS

M[r] GUERROUMI FAWZI
Centre de Développement des Technologies Avancées
128, chemin Mohamed Gacem B. P. 245 El-Madania Alger - ALGERIE
Fax: (213) 02 66-26-89 Tel: (213) 02 67-73-25

Résumé: Dans cet article, nous nous intéressons à une classe particulière de systèmes informatiques, *les systèmes réactifs* qui regroupent ceux dont l'objectif est plutôt le maintien d'une interaction avec un environnement que le calcul d'un résultat à partir de données en entrée. Pour énoncer la correction de ces systèmes, les formalismes de la logique classique ne peuvent être utilisés: en effet, il est nécessaire d'exprimer des propriétés portant sur le déroulement même de leur exécution. Il est donc légitime de les étudier à l'aide de techniques déductives formalisant des concepts temporels. Le propre des logiques "modales" dont fait partie la logique temporelle est de proposer de telles méthodes. Nous présentons ici un environnement de programmation logique mettant en oeuvre ce type de méthodologie. Le modèle de calcul sous-jacent considéré est celui des réseaux de Petri à prédicats.
Mots clés: vérification, réseaux de Petri, modèle PIPN, logique modale, logique temporelle arborescente CTL, programmation logique.

Abstract: most of parallel computing systems are mainly characterised by interaction with their environment. So they are called *reactive systems*. This paper is concerned with the correctness of such systems. A formal logic language can be proposed for the specification of sequential programs. This methodology cannot be used here: we have no means to express temporal properties characterising semantics of parallel programs. The *temporal logic* formalism, which formalises deductive techniques to deal with temporal concepts, is more appropriate for our problem. In this paper we present a logic programming environment for the verification of temporal logic specifications on Petri nets based models.
Keywords: verification, reactive systems, Petri nets, PIPN model, modal logics, branching time temporal logic CTL, logic programming.

I. INTRODUCTION

De nos jours, le parallélisme intervient à peu près partout en informatique, que ce soit un parallélisme réel, comme dans les machines multiprocesseurs et les réseaux d'ordinateurs, ou un parallélisme virtuel, comme celui qui existe entre les processus s'exécutant "simultanément" sur un ordinateur monoprocesseur. Or, la programmation de tels systèmes dans lesquels il y a des interactions et des synchronisations entre les divers processus, ou entre un processus et son environnement, est d'une complexité sans commune mesure avec celle des systèmes séquentiels classiques. Il est donc nécessaire de pouvoir vérifier formellement qu'un programme parallèle satisfait ses spécifications.

La plupart des systèmes parallèles sont conçus pour maintenir une interaction avec le monde extérieur plutôt que pour calculer des résultats; Pnuelli **[Pnu 86a]** les qualifie de systèmes *réactifs*, et il n'est pas facile de définir et de contrôler pleinement ces interactions. La combinatoire peut en effet être extrêmement complexe, et une variation minime dans un composant du système peut entraîner des modifications profondes dans son comportement global. De plus, la compré-

hension des phénomènes parallèles nécessite de prendre en compte un contrôle démultiplié et distribué.
La logique des programmes, un des domaines importants de l'informatique théorique, repose sur un système axiomatique permettant d'exprimer par des règles de déductions formelles les instructions d'un programme **[Hoa 69, Salwicki 70, Pratt 76, Harel 86]**. Ces logiques sont conçues pour l'étude des programmes séquentiels. Plus particulièrement elles sont employées pour la validation d'algorithmes et l'étude de la sémantique des langages de programmation. Il est donc naturel d'étendre cette méthodologie à l'étude des programmes parallèles à l'aide de techniques déductives formalisant des concepts temporels. La *logique temporelle* permet de répondre à de telles exigences.

Une fois choisie une logique temporelle comme formalisme de spécification, il est alors possible de définir des méthodes de vérification automatique (*model-checking*). Dans cet article nous construisons des procédures de *model-checking* pour la logique temporelle CTL, c'est-à-dire des algorithmes permettant de vérifier si, étant donné un système S et une spécification P, S satisfait P.

II. MODELE SEMANTIQUE DE LOGIQUE TEMPORELLE

En pratique, on associe à chaque système réactif, décrit par exemple dans un langage de programmation, un modèle sémantique qui représentera son comportement. Les modèles que nous considérons dans cette étude sont les réseaux de Petri à prédicats **[Gen 87]**. Les réseaux de Petri (RdP) **[BRAMS 83, Pet 81]** constituent un modèle fondamental pour la spécification du *parallélisme* qui caractérise un ensemble de tâches concurrentes. Cependant, les RdP se sont trouvés rapidement limités à cause de la banalisation des jetons entraînant ainsi une explosion combinatoire du graphe des marquages accessibles. Dans le modèle appelé **PIPN** (« Prolog Interpreted Petri Nets »), un jeton dans une place est structuré sous la forme de prédicat dans la logique du premier ordre. Pour un marquage donné, l'instanciation des jetons dans chaque place se fait alors par unification. Le caractère générique du modèle est exploité par l'introduction d'un champ sélecteur dans chaque transition. De plus, pour favoriser la réutilisation d'une même transition, une transition est attribuée par des clauses de Horn permettant de sélectionner les jetons lors du franchissement de cette dernière.

II.1 DEFINITION DU MODELE

Un **modèle PIPN** est un quintuplet *(P, T, tr, init, I)* où:

- P est un ensemble de prédicats appelés places
- T est un ensemble de prédicats appelés événements
- *tr* est un ensemble de quadruplets (t, pre, post, cond) appelés transitions où
  - t ∈ T est un prédicat appelé nom de la transition
  - pre (resp. post) est un ensemble non vide de prédicats appelés préconditions (resp. postconditions) de la transition
  - cond (champ sélecteur) est une séquence de prédicats appelés conditions sur les données. Ces prédicats réalisent une fonction de filtrage permettant de sélectionner les jetons lors du franchissement d'une transition.
- *init* est un ensemble non vide de prédicats de base (prédicats dont tous les termes sont des constantes) appelé état initial

• I est une fonction d'interprétation qui associe à la séquence *cond* des conditions sur les données une valeur de vérité.
Un marquage M d'un modèle PIPN est un ensemble de prédicats de base.

II.2 REGLE DE FRANCHISSEMENT

a) Transition sensibilisée
Si E est un ensemble de prédicats et $\theta = \{x_1=e_1, x_2=e_2, \ldots, x_n=e_n\}$ une **substitution**, E/θ représente la valeur de E après les remplacements des variables x_k par les termes e_k.
Une transition tr=(t, pre, post, cond) est sensibilisée par un marquage M SSI il existe une substitution θ telle que l'ensemble de ses préconditions pour cette substitution est un sous-ensemble de M et l'interprétation de la séquence des conditions sur les données retourne la valeur vrai:
$\exists\ \theta$ telle que $pre/\theta \subseteq M$ et $I(cond\theta)$ = vrai.

b) Franchissement d'une transition
Règle de franchissement: Soit M un marquage sensibilisant la transition tr=(t, pre, post, cond). Le marquage M' est le résultat du franchissement de la transition tr SSI:
$\exists\ \theta$ telle que $pre/\theta \subseteq M$ et $I(cond\theta)$ =vrai et
$M' = M - pre/\theta \cup post/\theta$

III. LANGAGE DE DESCRIPTION D'UN MODELE PIPN

III.1 DEFINITION DU LANGAGE

Un modèle PIPN est défini par l'ensemble de ses transitions, suivi du marquage initial et d'une interprétation des conditions sur les données.
La définition d'une transition est de la forme:

transition <nom de la transition>
pre: <ensemble de prédicats définissant les préconditions>
post: <ensemble de prédicats définissant les postconditions>
cond: <séquence de prédicats définissant les conditions sur les données>

La définition du marquage initial a la forme suivante:
init: <ensemble de prédicats de base définissant l'état initial>
L'interprétation des conditions sur les données se fait en termes de clauses de Horn **[CON 86]**.

Syntaxe du langage

<modèle>::= <transition>+
<init>
<conditions_données>*
transition::= transition <identificateur>
pre: <prédicat>+
post: <prédicat>+
cond: <prédicat>*
<init>::=init:<prédicat>+
<conditions_données>::= <clause de Horn>

III.2 INTERPRETATION PAR CLAUSES DE HORN

Chaque élément syntaxique est un fait ou une clause, décrit dans la syntaxe PROLOG. Un modèle PIPN constitue un programme PROLOG qui est directement simulé via une procédure de tir PROLOG. La forme externe du langage coïncide avec sa représentation interne dans l'outil.

A) INTERPRETATION DES TRANSITIONS

Convention: dans ce qui suit, un identificateur préfixé par le caractère « * » dénote un symbole de variable dans la syntaxe des clauses de Horn.

Une transition tr=(t, pré, post, cond) est interprétée par une clause de Horn sans condition de la forme:

tr(*t, pre(*pre), post(*post), cond(*cond)).

où les ensembles de préconditions (resp. postconditions) ainsi que les conditions sur les données sont interprétés par des listes de préconditions, de postconditions et de conditions sur les données respectivement.

Exemple: nous interpréterons la transition définie par la syntaxe suivante

```
trans   send(*site_i)
pre:    idle(*site_i)
post:   request(*site_i, *site_j)
        wait(*site_i)
cond:   dest(*site_i, *site_j)
```

par la clause de Horn sans condition:

tr(send(*site_i),
pre(idle(*site_i).nil),
post(request(*site_i,*site_j).wait(*site_i).nil),
cond(dest(*site_i,*site_j).nil)).

Le marquage initial est interprété par une liste de prédicats.

Exemple: nous interpréterons le marquage initial défini par la syntaxe suivante

init: idle(processus1) idle(processus2)

par la clause de Horn sans condition:

init(idle(processus1).idle(processus2).nil).

B) INTERPRETATION DE LA REGLE DE FRANCHISSEMENT

Une transition tr(*transition, pre(*pre), post(*post), cond(*cond)) est franchissable à partir d'un marquage *m SSI il existe une substitution θ telle que:

a) l'ensemble des préconditions pour la substitution θ est inclus dans le marquage *m: *pre/θ $\subseteq$ *m
b) l'interprétation des conditions sur les données retourne la valeur vrai: I(*condθ)= vrai.

Pour la substitution θ, le franchissement de la transition consiste alors à:

* retirer du marquage *m les préconditions de la transition
* ajouter au marquage intermédiaire les postconditions.

Le franchissement de la transition *transition donne le marquage *m' défini par l'équation:
*m' = *m - *pre/θ ∪ *post/θ
Le franchissement d'une transition défini par l'équation ci-dessus est interprété par la règle d'inférence:

```
tir(*m,*t,*m')- tr(*t,pre(*pre),post(*post),cond(*cond))
                - interprete(*cond)
                - enlever(*pre,*m,*m'')
                - inserer (*post,*m'',*m').
```

Les prédicats utilitaires **enlever** et **inserer** sont définis comme suit:
1. **enlever(*l1, *l2, *l)** est vrai si en enlevant les éléments de la liste *l1 à la liste *l2 on obtient la liste *l.
2. **inserer(*l1, *l2, *l)** est vrai si l'insertion des eléments de la liste *l1 à la liste *l2 donne la liste *l.

Le prédicat **interprete(*cond)** sert à interpréter les conditions sur les données. Son interprétation est donnée par la règle suivante:

```
interprete(nil).
interprete(*cond.*q):- *cond
                       - interprete(*q).
```

IV. SEMANTIQUE ET VALIDATION DE PROGRAMMES PARALLELES

IV.1 POURQUOI LA LOGIQUE TEMPORELLE

Le nécessaire avènement du calcul parallèle a ajouté une nouvelle dimension à la définition de systèmes de logiques adéquats: celle de leur incorporer la notion du temps. Pour traiter du concept supplémentaire de relations temporelles entre des événements, il faut définir des opérateurs permettant de "lier" des variables propositionnelles p aux instants t. La logique temporelle est un système formel de logique répondant à de telles exigences. Elle représente le parallélisme explicitement en décrivant le comportement d'un système au moyen d'une relation de causalité entre les occurrences de processus parallèles **[LAMP 78, LAMP 83]**. Tout programme peut être représenté par son *graphe de contrôle*. Une séquence d'exécution dans ce type de modèle est quasiment isomorphe à un modèle de la logique temporelle: Il suffit pour cela d'identifier "état du système" et "instant", le temps étant rythmé par le déclenchement des transitions. Une fois constatée cette analogie on peut alors trouver un codage approprié des propriétés de programmes dans le langage de la logique temporelle.

On distingue deux grandes classes de logique temporelle selon la structure du temps considérée:
1. logique temporelle arborescente (ou ramifiée) **[BENA 83, Queil 81, CLAR 81, Emer 83, Lich 84]**
2. logique temporelle linéaire **[Pnue 86b, Sis 82]**

La logique du temps linéaire correspond à la nature des programmes déterministes, tandis que la logique du temps arborescent est le formalisme adéquat pour l'étude des programmes non-déterministes **[Lam 80]**.
Comme la plupart des systèmes réactifs sont non-déterministes, nous ne considérons dans cet article que la logique temporelle arborescente et plus particulièrement la logique temporelle arborescente CTL (Conditonal Temporal Logic). Cette dernière spécifie des contraintes temporelles telles que:

« *il est toujours vrai que ...* »,
« *il existe un instant dans le futur où ...* »,
« *il a été vrai, au moins une fois dans le passé, que ...* »,
« *P doit être toujours vrai, jusqu'à ce que ...* »,
« *P était toujours vrai, depuis que ...* »,
. . .

IV.2 SEMANTIQUE DE LA LOGIQUE TEMPORELLE ARBORESCENTE CTL

La suite des états d'un programme est interprétée comme un modèle de la logique CTL constitué des éléments suivants:

a) S: un ensemble fini d'*états*. Chaque état représente une configuration de l'ensemble du système
b) Init: le *prédicat initial*. Il caractérise l'état initial du système
c) T: un ensemble fini de *transitions*.
d) R: une relation d'ordre binaire partielle.
$\forall$ s, s' $\in$ S; (s , s') $\in$ R SSI s' est un état accessible à partir de s par une transition t.

L'ensemble F des formules de logique CTL contient la logique du premier ordre étendue par les opérateurs temporels ALL(p , q), SOME(p , q), POT(p, q) et INEV(p , q). Si p et q $\in$ F alors 7p, p $\wedge$ q, p V q, p $\Rightarrow$ q, ALL(p , q), SOME(p , q), POT(p, q) et INEV(p , q) sont des formules de F. Les opérateurs temporels ALL(p , q), SOME(p , q), POT(p, q) et INEV(p , q) sont définis récursivement par les équations suivantes:

A) OPERATEURS DE BASE

s $\models$ PRE f	SSI	$\exists$ s' (s , s') $\in$ R et s' $\models$ f	(1)
s $\models$ PRETILDA f	SSI	$\forall$s' (s , s') $\in$ R $\Rightarrow$ s' $\models$ f	(2)

f étant une formule de logique CTL. s $\models$ f signifie que la formule f est satisfaite à l'état s du modèle.

B) DEFINITION DES OPERATEURS TEMPORELS

p , q $\in$ F:

s $\models$ ALL(p , q)	SSI	s $\models$ q $\wedge$ (7 p $\vee$ PRETILDA ALL(p , q))	(3)
s $\models$ SOME(p , q)	SSI	s $\models$ q $\wedge$ (7 p $\vee$ PRE SOME(p , q))	(4)
s $\models$ POT(p , q)	SSI	s $\models$ q $\vee$ (p $\wedge$ PRE POT(p , q))	(5)
s $\models$ INEV(p , q)	SSI	s $\models$ q $\vee$ (p $\wedge$ PRETILDA INEV(p , q))	(6)

V. INTERPRETATION DES OPERATEURS DE LA LOGIQUE CTL

Nous utilisons la programmation logique dans la définition des algorithmes de *model-checking*. s ⊨ f est interprétée par la résolution d'une règle d'inférence f(l , s). Cette dernière est évaluée récursivement sur un chemin d'origine l' état s dans le graphe des marquages accessibles dans un modèle PIPN. l est une trace d'états de ce chemin où le prédicat f(l , s) a été évalué. Cette liste d'états l permet au processus de résolution récursif d'arrêter l'évaluation de la règle sur un circuit du graphe équivalent au modèle de la logique CTL.

La relation d'accessibilité (s , s') ∈ R est interprétée par la clause de Horn sans condition arc(*s , *t , *s'). La variable formelle *t dénote la transition du modèle PIPN correspondant au franchissement du marquage *s au marquage *s'.

V.1 INTERPRETATION DES OPERATEURS DE BASE

Interprétation de l'équation (1):

```
PRE(*f,*l,*s):- arc(*s,*t,*s')
                - *f(*l,*s') - //.
```

L'opérateur prédéfini « // » (coupe choix) arrête la résolution à la première solution s' telle que s' ⊨f.

L'équation (2) est équivalente à:

s ⊨ PRETILDA f SSI $\top$ (illegible) s = CPRETILDA f

s ⊨ PRETILDA f SSI ¬ s ⊨ CPRETILDA f (2.1)

s ⊨ CPRETILDA f SSI ∃ s' (s , s') ∈ R et ¬ s' ⊨ f (2.2)

Interprétation des équations (2.1) et (2.2):

```
PRETILDA(*f,*l,*s):-NOT(CPRETILDA(*f,*l,*s)).
CPRETILDA(*f,*l,*s):- arc(*s,*t,*s') - NOT(*f(*l,*s')).
```

V.2 INTERPRETATION DES OPERATEURS TEMPORELS

s ⊨ POT(p , q) ⇐ s ⊨ q (5.1)

s ⊨ POT(p , q) ⇐ s ⊨ p ∧ PRE POT(p , q) (5.2)

```
POT(*p,*q,*l,*s):- out(*s,*l) - *q(*s) - //.
POT(*p,*q,*l,*s):- out(*s,*l) - *p(*s)
                  - PRE(POT(*p,*q),*s.*l,*s).
```

out(*s,*l) est vrai si la variable *s n'est pas un élément de la liste *l.

s ⊨ ALL(p , q) ⇐ s ⊨ q ∧ ¬p (3.1)

s ⊨ ALL(p , q) ⇐ s ⊨ q ∧ PRETILDA ALL(p , q) (3.2)

```
ALL(*p,*q,*l,*s):- in(*s,*l) - //.
ALL(*p,*q,*l,*s):- *q(*s) - NOT(*p(*s)) -//.
ALL(*p,*q,*l,*s):- *q(*s)
                  - PRETILDA(ALL(*p,*q),*s.*l,*s).
```

in(*s,*l) est vrai si la variable *s est un élément de la liste *l.

$s \models INEV(p,q) \Leftarrow \quad s \models q$ (6.1)

$s \models INEV(p,q) \Leftarrow \quad s \models p \wedge PRETILDA\ INEV(p,q)$ (6.2)

```
INEV(*p,*q,*l,*s):- out(*s,*l) -*q(*s) - //.
INEV(*p,*q,*l,*s):- out(*s,*l) - arc(*s,*t,*s') - // - *p(*s)
                  - PRETILDA(INEV(*p,*q),*s.*l,*s).
```

$s \models SOME(p,q) \quad \Leftarrow \quad s \models q \wedge \neg p$ (4.1)

$s \models SOME(p,q) \quad \Leftarrow \quad s \models q \wedge PRE\ SOME(p,q)$ (4.2)

```
SOME(*p,*q,*l,*s):- in(*s,*l) - //.
SOME(*p,*q,*l,*s):- *q(*s)
                  - NOT(arc(*s,*t,*s')) - //.
SOME(*p,*q,*l,*s):- *q(*s) - NOT(*p(*s)) - //.
SOME(*p,*q,*l,*s):- *q(*s)
                  - PRE(SOME(*p,*q),*s.*l,*s).
```

V.3 SIMPLIFICATIONS

POT(*p,*q,*s):- POT(*p,*q,nil,*s).
On attribue la valeur nil à la variable formelle *l.

Une deuxième simplification est l'élimination du prédicat conditionnel p de l'opérateur temporel POT(p , q). Nous supposons dans ce cas que p est vrai en tout état s du graphe des marquages équivalent au modèle de la logique CTL ($\forall s \in S; s \models p$). L'opérateur non conditionnel POT(p) est interprété par la règle d'inférence:
POT(*p,*s):- POT(true,*p,*s).
true(*s).
Des simplifications similaires sont définies sur les autres opérateurs temporels.

VI. EXEMPLES DE PROPRIETES EXPRIMABLES A L'AIDE DE FORMULES DE LOGIQUE TEMPORELLE

Considérons un système formé de n processus D_i, pouvant accéder à une ressource commune R, et d'un processus G chargé de gérer cette ressource. Chaque D_i communique avec G au moyen de deux variables booléennes, d_i et a_i, selon le protocole suivant:

- la variable d_i est modifiée uniquement par D_i qui lui affecte la valeur **vrai** pour signifier qu'il demande l'accès à la ressource; et la valeur **faux** pour relâcher R.

- la variable a_i est modifiée uniquement par G; a_i vaut **vrai** si la ressource a été accordée à D_i, ce dernier ne l'ayant pas encore rendue au gestionnaire.

Nous pouvons alors *spécifier formellement* les comportements corrects du système en énonçant des propriétés souhaitables, exprimables par des *formules de logique temporelle*.

P_1 " la ressource est accordée à au plus un demandeur à la fois "
Soit init(*s) le prédicat qui dénote l'état initial. P_1 s'exprime alors par la formule:
init $\Rightarrow$ ALL $\neg (a_i \wedge a_j)$ $i \neq j$
Il s'agit d'une propriété d'exclusion mutuelle, exprimant l'invariance d'une certaine condition ($\neg (a_i \wedge a_j)$). De manière générale, les *propriétés d'invariance* s'exprimeront comme sur notre exemple à l'aide de l'opérateur ALL.

P_2 " Si le processus D_i demande la ressource, il l'obtiendra "
s'exprime par la formule:
init $\Rightarrow$ ALL($d_i \Rightarrow$ INEV a_i)

P_3 " Si le processus D_i obtient la ressource, il la relâchera "
s'exprime par la formule:
init $\Rightarrow$ ALL($a_i \Rightarrow$ INEV $\neg d_i$)

P_2 et P_3 sont des propriétés de *vivacité*. Ces propriétés s'expriment en général à l'aide d'un opérateur INEV.

VII. CONCLUSION

Nous avons montré dans cet article comment la logique temporelle pouvait être un bon outil pour la preuve de propriétés des programmes. Nous pouvons remarquer que tout système à nombre fini d'états (en l'occurrence le graphe des marquages d'un réseau de Petri borné) définit un modèle " canonique " de logique temporelle arborescente (l'arbre de tous les calculs possibles); par conséquent, pour prouver une propriété, exprimable par une formule de logique temporelle, il suffit de montrer que la formule est *satisfaite dans ce modèle*. Notre article est consacré à une telle approche.
Nous avons donc implémenté un environnement de programmation logique mettant en oeuvre une telle méthodologie. A partir d'un programme défini par l'utilisateur, sous la forme d'un modèle PIPN (le modèle de calcul sous-jacent est celui des réseaux de Petri à prédicats), notre outil calcule le graphe de transitions des états globaux du programme. Des variables propositionnelles définies par l'utilisateur permettent de définir des propriétés élémentaires de ces états, telles que la présence du contrôle en tel ou tel point. L'utilisateur peut alors demander l'évaluation de formules temporelles représentant des propriétés complexes de son programme.
Les perspectives de développement envisagées sont principalement l'extension de l'interpréteur des formules de logique temporelle CTL. En effet celui-ci est incapable d'assister l'utilisateur au cas où une spécification formelle (formule de logique temporelle interprétée) ne serait pas vérifiée sur l'automate à états finis équivalent au modèle de son application. Dans la mesure du possible, cette aide pourrait avoir la forme d'une séquence de transitions, dont le tir fait apparaître un contre-exemple dans la satisfaction de la propriété en question.

BIBLIOGRAPHIE

Bak 83 J. W. Bakker, J. A. Bergstra, J. J. C. Klopp
Linear time and branching time semantics for recursion with merge.
Proc. ICALP 83, LNCS, #154, pages 39-51, Springer-Verlag, 1983

BENA 83 M. Ben-Ari, A. Pnuelli, Z. Manna
The temporal logic of branching time.
Acta Informatica, volume 20, 1983

BRAMS 83 G. W. BRAMS
Réseaux de Petri: théorie et pratique.
Masson 1983

CLAR 81 E. M. Clarke, E. A. Emerson
Design and synthesis of synchronization skeletons using branching time temporal logic.
Proc. of an IBM Workshop on Logic of Programs, LNCS, #131, pages 51-71, Springer-Verlag, 1981

CON 86M. Condillac
PROLOG fondements et applications.
Editions Dunod, 1986

Emer 83 E. A. Emerson, J. Y. Halpern
'sometimes' and 'not never' revesited: on branching versus linear time.
11th Symposium on Principles Of Programming Languages, Austin,Texas, pages 127-140, 1983

Gen 87 H.J Genrich
Predicate/Transition nets.
Draft, 1987

Harel 86 D.Harel
Dynamic logic.
Handbook of philosophical logic, (Gabbay D., Guenthner F. Eds), Vol. III, Reidel, 1986

Hoa 69 C.A.R Hoare
An axiomatic basis for computer programming.
Comm. of the ACM, 12, pp 576-583, 1969

LAMP 78 L. Lamport
Time, clocks and the ordering of events in distributed systems.
Communications of the ACM, vol 21, 7 juillet 78

LAMP 80 L. Lamport
'sometimes' is sometimes 'not never': on the temporal logic of programs.
Proc. 7th ACM Symposium on POPL, pages 174-185, january 1980

LAMP 83 L. Lamport
What good is temporal logic?
Proc. IFIP 83, pages 657-668, North-Holland, 1983

Lich 84 O. Lichtenstein, A. Pnuelli
Checking that finite state concurrent programs satisfy their linear specification.
12th Symposium on Principles of Programming Languages, Austin, Texas, pages 97-107, 1984

Pet 81 J. Peterson
Petri net theory and the modelling of systems.
Prentice Hall, 1981

Pnue 77 A. Pnuelli
The temporal logic of programs.
Proc. of the 18th Symposium on the Foundations of Computer Science, ACM, Nov. 1977

Pnue 85 A. Pnuelli
Linear and branching structures in the semantics and logics of reactive systems.
12th ICALP, LNCS, pages 15-32, Springer-Verlag, 1985

Pnue 86a A. Pnuelli
Specification and development of reactive systems.
In Proc. IFIP'86, pp 845-858, North Holland, 1986

Pnue 86b A. Pnuelli
Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends.
LNCS #224, Current trends in concurrency, pages 510-584, Springer-Verlag, 1986

Pratt 76 Pratt V.R
Semantical considerations on Floyd-Hoare logic.
Proc. of the 17th Symp. on Found of Computational Science, pp 109-121, 1976

Queil 81 J. P. Queille, J. Sifakis
Specification and verification of concurrent systems in CESAR.
Proc. 5th International Symposium in Programming, 1981

Salwicki 70 Salwicki A.
Formalized algorithm language.
Bull. Ac. Pol. Sc., 18(5), pp 227-232, 1970

Sis 82 A. P. Sistla, E. M. Clarke
The complexity of propositionnal linear temporal logics.
10th Symposium on Principles of Programming Languages, Austin, Texas, pages 159-168, 1982