Abstractions booléennes pour la vérification des systèmes temps-réel

Eun-Young Kang

▶ **To cite this version:**

Eun-Young Kang. Abstractions booléennes pour la vérification des systèmes temps-réel. Autre [cs.OH]. Université Henri Poincaré - Nancy 1, 2007. Français. NNT : 2007NAN10089 . tel-01748252

HAL Id: tel-01748252
https://hal.univ-lorraine.fr/tel-01748252

Submitted on 29 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

Nancy-Université

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Abstractions booléennes pour la vérification des systèmes temps-réel

THÈSE

présentée et soutenue publiquement le 8 Novembre 2007

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Eun-Young Kang

Composition du jury

Rapporteurs :	Christian Attiogbé	Maître de conférence
	Jean-Paul Bahsoun	Prof. Université Paul Sabatier France
Examinateurs :	Jean-Paul BODEVEIX	Prof. Université Paul Sabatier France
	Isabelle CHRISMENT	Prof. UHP Nancy 1 France
	Yamine AIT-AMEUR	Prof. LISI-ENSMA France
	Dominique MÉRY	Prof. UHP Nancy 1 France
	Stephan MERZ	Directeur de recherche INRIA France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria

Mis en page avec la classe thloria.

Abstract

This thesis provides an efficient formal scheme for the tool-supported real-time system verification by combination of abstraction-based deductive and *model checking* techniques in order to handle the limitations of the applied verification techniques. This method is based on IAR (Iterative Abstract Refinement) to compute finite state abstractions. Given a transition system and a finite set of predicates, this method determines a finite abstraction, where each state of the abstract state space is a true assignment to the abstraction predicates. A theorem prover can be used to verify that the finite abstract model is a correct abstraction of a given system by checking conformance between an abstract and a concrete model by establishing/proving that a set of verification conditions are obtained during the IAR procedure. Then the safety/liveness properties are checked over the abstract model. If the verification condition holds successfully, IAR terminates its procedure. Otherwise more analysis is applied to identify if the abstract model needs to be more precise by adding extra predicates. As abstraction form, we adopt a class of predicate diagrams and define a variant of predicate diagram PDT (*Predicate Diagram for Timed systems*) that can be used to verify real-time and parameterized systems.

Key words : Real-time system verification, model checking, deductive technique, and predicate abstraction

Résumé

Cette thèse présente un schéma formel et efficace pour la vérification de systèmes temps-réel. Ce schéma repose sur la combinaison par abstraction de techniques déductives et de *model checking*, et cette combinaison permet de contourner les limites de chacune de ces techniques. La méthode utilise le raffinement itératif abstrait (IAR) pour le calcul d'abstractions finies. Etant donné un système de transitions et un ensemble fini de prédicats, la méthode détermine une abstraction booléenne dont les états correspondent à des ensembles de prédicats. La correction de l'abstraction par rapport au système d'origine est garantie en établissant un ensemble de conditions de vérification, issues de la procédure IAR. Ces conditions sont à démontrer à l'aide d'un prouveur de théorèmes. Les propriétés de sûreté et de vivacité sont ensuite vérifiées par rapport au modèle abstrait. La procédure IAR termine lorsque toutes les conditions sont vérifiées. Dans le cas contraire, une analyse plus fine détermine si le modèle abstrait doit être affiné en considérant davantage de prédicats. Nous identifions une classe de diagrammes de prédicats appelés PDT (*predicate diagram for timed system*) qui décrivent l'abstraction et qui peuvent être utilisés pour la vérification de systèmes temporisés et paramétrés.

Mots clés : vérification de systèmes temps-réel, model checking, technique déductive, et prédicats d'abstraction

Abstract

Model checking has been broadly considered and useful in verification for finite state real time systems. Theoritically optimal algorithms make model checking successful in a way that a system and a property to check and automatically either proves it correct or comes up with a violation. Unfortunately, although these algorithms are fully automatic, they are confronted with a state explosion problem. They are typically exponential in the number and maximum values of clocks.

On the other hand, deductive techniques can in principle be used to verify infinite, or unbounded systems and also large finite state systems, based on suitable sets of axioms and inference rules. Although they are supported by theorem provers and interactive proof assistants, their use requires considerable expertise and tedious/repetitive user interaction.

Predicate abstraction provides a different approach to computing finite state abstractions. Given a transition system abd a finite set of predicates, this method determines a finite abstraction, where each state of the abstract state space is a true assignment to the abstraction predicates.

Model checking and abstraction/deductive techniques are therefore complementary. Thus we have proposed an efficient scheme by combination of abstraction-based deductive and model checking techniques that should give a rise to powerful verification environment. This method is based on our IAR (Iterative Abstract Refinement) algorithm to provide a formal framework for verifying a rich class or safety and liveness properties of timed systems.

IAR computes an abstract model and a theorem prover can be used to verify that the finite abstract model is a correct abstraction of a given system by checking conformance between an abstract and a concrete model by establishing/proving that a set of verification conditions are obtained during the IAR procedure. Then the safety/liveness properties are checked over the abstract model. If the verification condition holds successfully, IAR terminates its procedure. Otherwise more

analysis is applied to identify if the abstract model needs to be more precise by adding extra predicates.

As abstraction form, we adopt a class of predicate diagrams and define a variant of predicate diagram PDT (Predicate Diagram for Timed systems) that can be used to verify real-time and parameterized systems.

Eun-Young Kang

Acknowledgements

The road has been rather long — not to mention somewhat winding.

Over the years it has been my good fortune to encounter many people who have given me their time, companionship, professional and personal help, and above all, their patience. This was certainly more than was warranted by my seeming determination to indefinitely position the deadline for finishing this thesis at "this year".

I would first of all like to thank my mentor, Stephan Merz. He not only gave me the scientific support and supervision that a graduate student can expect from his professional research, but he also allowed and encouraged me to continue my Ph.D work as part of "Cotutelle", in the MOSEL group at LORIA after I had formally left the Netherlands. My sincere thanks to him : I could not have wished for a more thorough discussion partner and sounding board (on *any* subject under the sun). During the hardest days of my research his amazing knowledge (particularly with respect on logics and formal methods) has been an excellent guiding light, helping the methodology described in this thesis to come to life. Without his great supporting, I would never have made it this far.

Hans Toetenel, then of Delft University in Holland, was my primary supervisor in the early years. His ideas, his research, and especially his unique brand of enthusiasm form the model checking on which much of this thesis is built. I cannot help but feel apologetic towards him over the fact that the specification language — so very much a cornerstone of our work in those early days — eventually fell by the wayside...

There are five fellow researchers whom I would specifically like to thank for the support they have given me over the years : Ronald Lutje Spelberg for his work on the specification notation with formal semantics of XTGs, Dominique Cansell, Dominique Mery and Stephan Merz for the early work on the definition and semantics of predicate diagrams, and Loic Fejoz for his implementation of

predicate diagrams in the DIXIT toolkit.

The list of all the co-workers, group members, and music band members that I have worked, talked, had lunch with, and played with over the years is to long for individual gratifications. My gratitude goes out to all these colleagues at LORIA-INRIA, and former colleagues at Delft University ; as well as to my rock band "mumbling-of-goldfishes" in Nancy/Strasbourg.

A special word of gratitude, finally, to all the members of projects that I've been involved with and their follow-up efforts and spin-offs. The projects are finished, and for most of us our ways already parted (long ago), but they have been stimulating and exciting times, and I treasure the memories. I am grateful to all the people that I knew during my stay in Holland and in France. They have truly made my stay in Europe a very pleasant one.

Moving towards more personal acknowledgements, I would like to thank all my family and friends[1] abroad for supporting me — with a special shout-out to my *best companions ever* Andrea and Jules — for your help, friendship and patience, and for the fact that you never gave in to the temptation to make fun of my "being-floating-all-around-the-world" situation. Well — hardly ever.

I am, of course, particularly indebted to my parents for their monumental, unwavering support and encouragement, despite the distance. They have truly always been there for me, and without them none of this would have been even remotely possible.

Nancy,
September 2007

Eun-Young Kang

"Through the unknown, we'll find the new"
– Charles Baudelaire

[1] You know who you are.

Table des matières

Table des figures

Chapitre 1

Résumé

1.1 Introduction

Les techniques de *model checking*, fondées sur une exploration de l'espace d'état d'un système de transitions, sont désormais devenues ubiquitaires pour la vérification de circuits et de protocoles de communication. Dans ces applications, il est aisé de modéliser l'application par un système fini de transitions. Des travaux fondateurs par Alur et Dill, Henzinger et d'autres [3, 34] ont démontré que des techniques de *model checking* peuvent aussi être développées pour les systèmes temps réel ; les outils correspondants ont atteint un degré significatif de maturité et sont capables de traiter des systèmes non-triviaux [6, 61]. Les attraits principaux de l'approche par *model checking* sont, d'une part, l'entière automatisation de la vérification, et d'autre part, sa capacité de fournir des contre-exemples quand le système ne vérifie pas la propriété décrivant les comportements désirables.

Cependant, le *model checking* ne s'applique aux systèmes temps-réel que sous certaines restrictions. Notamment, le système doit être décrit par un automate temporisé dont l'espace d'états discret (hormis les valeurs des horloges) est fini. Cette restriction n'est généralement pas vérifiée lorsqu'on essaie de vérifier les systèmes logiciels. On construit alors des approximations *ad-hoc* qui seront validées par les outils de *model checking*. Une alternative consiste à utiliser des techniques déductives, fondées sur des ensembles d'axiomes et de règles, qui s'appliquent en principe à la vérification de systèmes infinis. Les démonstrateurs automatiques et assistants interactifs sont à la disposition de l'utilisateur pour l'aider à mettre en œuvre ces techniques déductives, mais leur utilisation nécessite une expérience importante et requiert souvent beaucoup d'étapes fastidieuses d'interaction [4].

Les techniques algorithmiques d'exploration d'états et les techniques déductives sont donc complémentaires et leur utilisation conjointe devrait conduire à des techniques et environnements puissants de vérification. Par exemple, on pourra

utiliser un assistant de preuve afin de démontrer qu'on modèle fini est une bonne abstraction du système donné, tandis que les propriétés de l'abstraction peuvent être établies en utilisant le *model checking*. Afin de concrétiser cette idée on identifiera un format adéquat qui servira d'interface entre les techniques déductives et algorithmiques de vérification, et on identifiera des conditions de vérification qui rentrent dans le champ d'application d'outils (de préférence automatiques) de déduction.

Le principe de l'abstraction booléenne [30, 59] est un principe largement reconnu pour la vérification de logiciels. Il est à la base d'outils tels que SLAM [10] et BLAST [36] qui en outre appliquent des algorithmes pour le raffinement d'abstractions lorsque le *model checker* fournit un contre-exemple pour le modèle abstrait qui ne peut être rejoué sur le modèle d'origine.

La présente thèse a pour objectif de développer une méthodologie, appelée IAR (*iterative abstraction refinement*, raffinement itérative d'abstractions), pour la vérification dans le contexte de systèmes temps-réel. Pour ce faire, nous nous plaçons dans un cadre d'abstraction et de raffinement qui permet de réduire le nombre d'états d'un modèle en éliminant des comportements qui ne sont pas essentiels pour la vérification. Les techniques génériques sont connues sous le nom de raffinement incrémental d'abstractions [2]. Dans notre cas nous proposons une classe de diagrammes appelée PDT (Predicate Diagrams for Timed systems) qui représentent les abstractions booléennes et qui nous permettent de vérifier les propriétés de sûreté et de vivacité de systèmes temporisés.

Nous montrons comment faire le lien entre les PDTs et les systèmes temps-réel représentés sous forme de graphes d'automates temporisés étendus (XTG), un formalisme combinant les automates temporisés et des contraintes permettant la modélisation de domaines infinis de données [45, 5]. Le PDT représentera une abstraction finie d'un XTG, et la correction de cette abstraction sera établie en démontrant un certain nombre de conditions de vérification écrites en logique de premier ordre. Aussi les techniques de *model checking* seront utilisées afin d'établir des propriétés de correction des PDT (exprimées en logique temporelle).

1.1.1 Les propos de la thèse

Cette thèse étudie trois sujets qui sont l'abstraction, la déduction et le *model checking* pour la vérification de systèmes. Nous allons supposer que le *model checking* sera utilisé pour la vérification de propriétés des modèles abstraits.

Avant d'entrer dans le vif du sujet nous commençons par donner une idée générale de notre objectif qui est d'utiliser de nouveaux concepts de vérification : le *model checking* à proprement parler s'effectue en deux phases bien distinctes, comme c'est illustré dans la figure 1.1.(a). D'abord, une description S d'un système doit être «dépliée» pour en obtenir un modèle C – nous appelons cela la *construction* (du modèle), formellement décrite par une fonction R appelée *représentation*. Puis la propriété ϕ qui nous intéresse doit être vérifiée sur ce modèle : $C \models \phi$.

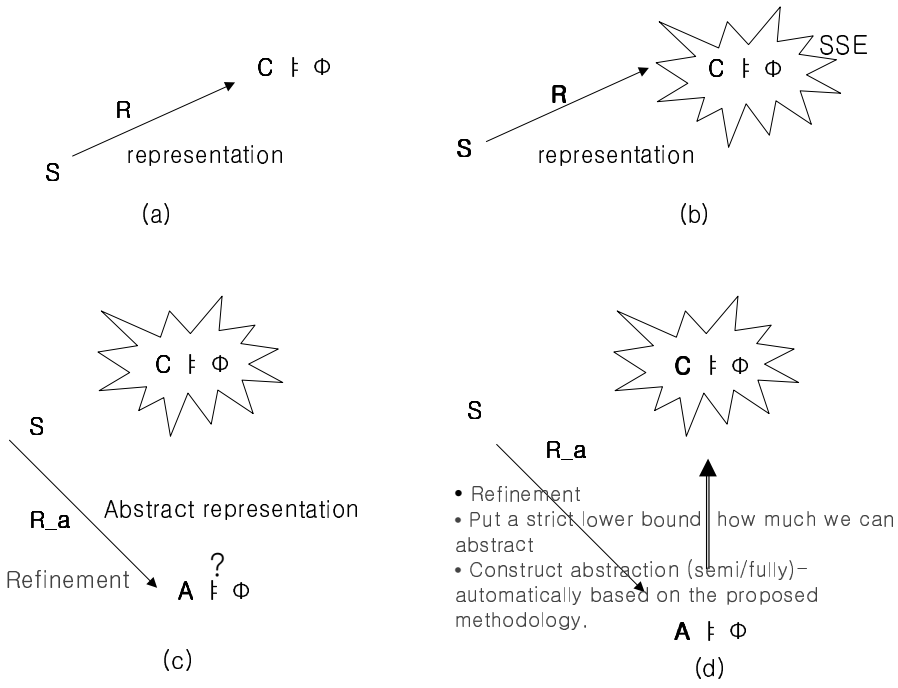


Fig. 1.1: Model checking *et abstraction.*

Il est bien connu que le problème de l'explosion combinatoire, illustré dans la figure 1.1.(b), est le facteur limitant de l'application de cette méthode.

Nous préférions disposer d'une fonction directe permettant de construire le modèle abstrait A à partir de la description S du système, et nous appelons cette fonction la *représentation abstraite* R_a.

Comme nous l'avons dit plus haut, on peut faire abstraction des états des comportements du modèle concret qui ne sont pas essentiels pour la vérification de façon à ce que la taille du modèle soit réduite de manière importante. C'est illustré dans la figure 1.1.(c).

Nous aimerions ainsi réduire le modèle à une abstraction A sur laquelle une analyse efficace peut être effectuée, mais toutefois à condition que la satisfaction de ϕ sur A implique sa satisfaction sur C. On observera que nous ne demandons pas que des résultats négatifs soient aussi préservés : si A ne vérifie pas ϕ alors on ne saura rien dire sur C. Il est toujours possible que trop d'information ait été perdu dans la construction du modèle abstrait (voir Figure 1.1.(d)).

Pour résumer, notre objectif principal est de représenter un système complexe donné par une abstraction simple de façon à ce que le problème de l'explosion combinatoire soit évité car il n'est pas possible d'explorer tout l'espace d'états

étant donné des ressources limitées en temps et mémoire. Pour ce faire nous proposons le format des PDT qui permettent de décrire de façon concise mais vérifiable un modèle abstrait qui sera utilisé pour la vérification. Cette approche requiert donc une interaction limitée avec l'utilisateur humain. Le PDT peut être affiné afin d'éliminer des faux résultats négatifs. Nous appelons un tel PDT un modèle *complet*.

Les techniques permettant de construire A étant donné S sont centrales dans la théorie de la représentation abstraite. La première partie de cette thèse (chapitres 2 et 3) traitent de ces aspects. Le chapitre 2 présente des théories génériques pour la modélisation de systèmes. Le chapitre 3 donne un aperçu systématique de la théorie des représentations abstraites et décrit l'évaluation de propriétés. Une notion de représentations abstraites de systèmes (PDT : *predicate diagrams for timed systems*, diagrammes de prédicats pour les systèmes temps réel) est introduite afin de vérifier des propriétés exprimées en LTL (logique temporelle linéaire) du système concret en les vérifiant sur le PDT. Un PDT conforme au système concret et qui préserve les propriétés que nous désirons vérifier peut être considéré comme une représentation abstraite complète (un modèle complet) du système concret.

La construction d'un PDT complet est le sujet principal de cette thèse et le chapitre 4 décrit comment un tel PDT peut être obtenu dans le cadre de la méthodologie IAR, et aussi comment des propriétés de sûreté et de vivacité peuvent être vérifiées en se servant du PDT.

La deuxième partie de cette thèse étudie des applications des PDT. Le chapitre 5 montre que le formalisme des PDT peut être étendu à la vérification des systèmes paramétrés. Dans le chapitre 6 nous étudions davantage la génération d'un PDT qui peut être considéré comme une représentation abstraite complète du système étudié et nous montrons la pertinence de ce PDT en nous servant d'une étude de cas.

1.1.2 Les aspects clefs de cette thèse

Cette thèse est focalisée sur l'application de techniques d'abstractions et de leur raffinements (IAR) pour la vérification de systèmes temps réel. Cette technique est développée en considérant les trois aspects clefs qui sont la représentation abstraite, le raffinement d'abstractions et l'évaluation d'une représentation abstraite donnée. Bien que ces aspects ne puissent pas toujours être vus séparément, il est utile de les considérer comme étant des aspects différents d'un problème de vérification :

1. *Représentation abstraite :* étant donné une spécification d'un système, c'est à dire une description implicite d'un système «concret», IAR permet de représenter l'espace d'états du modèle concret d'une manière abstraite, en réduisant le nombre d'états et en faisant abstraction de comportements qui ne sont pas essentiels à la vérification.

2. *Raffinement d'abstractions :* IAR garantit que la représentation est toujours une abstraction correcte du modèle concret. Un outil de démonstration formelle est utilisé afin d'établir une correspondance entre les systèmes concret et abstrait. Bien que ces outils demandent souvent beaucoup d'expertise et un effort considérable d'interaction de la part de l'utilisateur, nous nous attendons à ce que les obligations de preuve soient relativement faciles pour le prouveur. Ainsi, l'approche promet d'être applicable à la vérification de systèmes infinis.
3. *Evaluation (vérification) :* le mécanisme devrait permettre à un outil de *model checking* de décider si la représentation abstraite doit être affinée en vérifiant des propriétés sur cette représentation.

La distinction entre ces aspects sera utile tout au long de la thèse. Pour chacun des aspects nous allons proposer une solution, toujours sous l'hypothèse fondamentale que la représentation abstraite est fondée dans le cadre IAR de raffinement d'abstractions. Ainsi, nous pouvons dire que notre objectif est de trouver des approches de représentation abstraite et d'évaluation appropriées à la vérification.

1.2 Modéliser les systèmes et leurs propriétés

1.2.1 Modélisation de systèmes : les XTG

Nous représentons un système temps réel sous forme d'un XTG (*extended timed automata graph*, graphe d'un automate temporisé étendu) [5, 56], un formalisme qui intègre les automates temporisés habituels, l'échange synchrone de valeurs entre processus parallèles et un langage pour la modélisation de données. La sémantique de XTG est définie en termes de structures temporisées, aussi connues sous le nom de systèmes de transitions temporisés.

Définition 1.1 (système de transitions temporisé)**.** *Un* système de transitions temporisé *ou* structure temporisée *est un triplet* $\langle S, S_0, T\rangle$ *où*

- S *est un ensemble d'états,*
- $S_0 \subseteq S$ *est le sous-ensemble d'états initiaux et*
- $T \subseteq S \times (\mathbb{R}^{\geq 0} \cup \{\mu\}) \times S$ *est la relation de transitions.*

Une exécution *d'une structure temporisée est une suite infinie*

$$\pi = s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} s_2 \ldots$$

où $s_0 \in S_0$ *est un état initial et où* $\langle s_i, \lambda_i, s_{i+1}\rangle \in T$ *est une transition pour tout* $i \in \mathbb{N}$*.*

Les systèmes de transitions temporisés ont deux sortes de transitions : celles correspondant au passage de temps sont étiquetées par un nombre réel non négatif qui indique combien de temps a passé. Les transitions discrètes correspondent à l'évolution de l'état et sont étiquetées par μ. Notre définition des XTG sera

paramétrée par un langage sous-jacent pour la modélisation d'états. N'ayant pas besoin de fixer une signature précise, nous nous plaçons dans le cadre syntaxique générique suivant :

Définition 1.2 (modèles de données)**.** *Un langage de modélisation de données fournit les domaines syntaxiques suivants :*
- *V : un ensemble fini de variables,*
- *$V_c \subseteq V$: un ensemble d'horloges,*
- *$Expr$: un ensemble d'expressions (sur l'ensemble V) dénotant des valeurs et*
- *$Bexpr \subseteq Expr$: l'ensemble d'expressions booléennes.*

De manière analogue, nous ne fixons pas de sémantique précise, mais demandons simplement l'existence d'un domaine sémantique adéquat et d'une fonction d'évaluation.

Définition 1.3 (environnement)**.** *Nous supposons un univers Val de valeurs comprenant l'ensemble $\mathbb{R}^{\geq 0}$ des nombres réels non négatifs et les valeurs booléennes tt et ff. Un environnement est une fonction $\rho : V \to Val$ associant des valeurs aux variables tel que $\rho(c) \in \mathbb{R}^{\geq 0}$ pour tous $c \in V_c$. Si ρ est un environnement et $\delta \in \mathbb{R}^{\geq 0}$ on dénotera par $\rho[+\delta]$ l'environnement où tous les horloges sont augmentés par δ :*

$$\rho[+\delta](v) = \begin{cases} \rho(v) + \delta & \text{si } v \in V_c \\ \rho(v) & \text{sinon} \end{cases}$$

Nous supposons donnée une fonction d'évaluation

$$[\![\_]\!]_\_ : Expr \to (V \to Val) \to Val$$

qui associe une valeur $[\![e]\!]_\rho$ à toute expression $e \in Expr$ et environnement ρ. Nous exigeons que $[\![e]\!]_\rho \in \{tt, ff\}$ pour toute expression $e \in Bexpr$.

Outre le langage précis de modélisation, notre définition suivante des XTG fait abstraction du modèle de parallélisme et ne permet que la modélisation des graphes simples.

Définition 1.4 (XTG)**.** *Un processus d'un XTG est donné par un sextuplet $\langle Init, L, l_0, I, E, U\rangle$ où*
- *le prédicat $Init \in Bexpr$ décrit la condition initiale (des variables) du processus,*
- *L est un ensemble fini d'états (de contrôle),*
- *$l_0 \in L$ est l'état initial,*
- *$I : L \to Bexpr$ associe un prédicat (l'invariant) à chaque état,*
- *$E \subseteq L \times Bexpr \times 2^{V \times Expr} \times L$ est un ensemble d'arêtes et $U \subseteq E$ est le sous-ensemble d'arêtes urgentes. Chaque arête est représentée par un quadruplet $\langle l, g, u, l'\rangle$ où :*
  - *$l \in L$ est l'état source,*

– *l'expression* $g \in Bexpr$ *représente la garde,*
– $u \subseteq V \times Expr$ *définit la mise à jour des variables et*
– $l' \in L$ *est l'état cible.*

Les mises à jour sont définies par des ensembles de couples $\langle v, e\rangle$ *d'une variable* v *et d'une expression* e *qui sera affectée à la variable. Chaque variable apparaîtra dans au plus un couple dans* u*.*

Un XTG est un ensemble fini de processus.

À tout XTG on associe une structure temporisée dont les états consistent en les états de contrôle actifs de l'XTG, ainsi qu'en un environnement pour les variables.

Définition 1.5 (sémantique d'XTG)**.** *Soit* $\mathcal{X}$ *un XTG comportant les processus* $P_1, \ldots, P_m$*. La structure temporisée* $\mathcal{T} = \langle S, S_0, T\rangle$ *générée par* $\mathcal{X}$ *est la structure telle que*

– S_0 *contient les tuplets* $\langle l_{10}, \ldots, l_{m0}, \rho\rangle$ *où* l_{i0} *est l'état initial du processus* P_i *et où* $[\![Init_i]\!]_\rho = tt$ *pour les conditions initiales* $Init_i$ *de tous les processus.*
– *Pour tout état* $s = \langle l_1, \ldots, l_m, \rho\rangle \in S$*, tout* $i \in \{1, \ldots, m\}$ *et toute arête* $\langle l_i, g, u, l_i'\rangle \in E_i$ *avec* $[\![g]\!]_\rho = tt$*, la structure* $\mathcal{T}$ *contient une transition* $\langle s, \mu, s'\rangle \in T$ *telle que* $s' = \langle l_1', \ldots, l_m', \rho'\rangle$ *avec* $l_j' = l_j$ *pour tout* $j \neq i$ *et*

$$\rho'(v) = \begin{cases} [\![e]\!]_\rho & si\ \langle v, e\rangle \in u \\ \rho(v) & sinon \end{cases},$$

à condition toutefois que $[\![I(l_j')]\!]_{\rho'} = tt$ *pour tout* $j \in \{1, \ldots, m\}$*.*
– *Pour tout état* $s = \langle l_1, \ldots, l_m, \rho\rangle \in S$ *et* $\delta \in \mathbb{R}^{\geq 0}$ *la structure* $\mathcal{T}$ *contient la transition* $\langle s, \delta, s'\rangle \in T$ *où* $s' = \langle l_1, \ldots, l_m, \rho[+\delta]\rangle$*, à la double condition que*
  – $[\![I(l_i)]\!]_{\rho[+\varepsilon]} = tt$ *pour tout* $i \in \{1, \ldots, m\}$ *et tout* $0 \leq \varepsilon \leq \delta$*, c'est à dire les invariants locaux de tous processus sont vérifiés pendant le passage de temps et*
  – *pour toutes les arêtes urgentes* $\langle l_i, g, u, l_i'\rangle$ *dont l'état source est un des états actifs dans* s*, la garde* g *est fausse pour tout* $0 \leq \varepsilon < \delta$*, c'est à dire que* $[\![g]\!]_{\rho[+\varepsilon]} = ff$*.*

Nous allons encore étendre les XTG par un concept de communication synchrone de valeurs entre processus. Pour ce faire nous commençons par introduire les systèmes d'XTG qui consistent en plusieurs XTG communiquant à travers des canaux de communication, puis réduire ces systèmes à des XTG de base.

Nous introduisons d'abord des primitives décrivant l'échange de valeurs.

Définition 1.6 (primitives de communication)**.** *Soit* $Lab_c = \{lab_{c1}, lab_{c2}, \ldots\}$ *un ensemble d'étiquettes de communication et* $\overline{Lab_c} = \{\overline{lab_{c1}}, \overline{lab_{c2}}, \ldots\}$ *un ensemble d'étiquettes complémentaires. L'échange de valeurs est représenté par un triplet* $\langle ch, ia, oa\rangle$ *où*

– $ch \in Lab_c \cup \overline{Lab_c}$ *identifie un canal de communication,*
– $ia = \langle v_1, \ldots, v_n\rangle$ *est un n-uplet (potentiellement vide) de variables et*

– $oa \in \langle e_1, \ldots, e_n \rangle$ *est un n-uplet (éventuellement vide) d'expressions.*

Nous dénotons par VP_V *l'ensemble d'expressions modélisant l'échange de valeurs formées à partir de l'ensemble* V *de variables. Deux étiquettes de communication sont complémentaires si l'un est la version surlignée de l'autre, comme* lab_c *et* $\overline{lab_c}$.

Nous utilisons la syntaxe concrète $lab_c?v_1?v_2 \ldots !e_1!e_2 \ldots$ au lieu de la représentation explicite $\langle lab_c, \langle v_1, v_2, \ldots \rangle, \langle e_1, e_2, \ldots \rangle \rangle$. Le plus souvent, il y a un transfert d'une seule valeur, ou d'aucune valeur lorsqu'il s'agit d'une synchronisation pure. La direction de transfert est indiquée par le nom de l'étiquette. Les expressions avec des étiquettes complémentaires peuvent synchroniser pour échanger des valeurs, comme par exemple $\overline{coffee}!3$ et $coffee?strength$ (où *strength* est une variable).

Définition 1.7 (systèmes étendus d'XTG)**.** *Un système étendu d'XTG est un triplet* $\mathcal{X} = \langle GInit, G, Ch \rangle$ *où*

– $GInit \in Bexpr$ *donne le prédicat initial global concernant (les variables de) l'ensemble des processus,*
– $G = \{gr_1, gr_2, \ldots\}$ *est un ensemble fini d'XTG,*
– $Ch : EE \to (VP_V \cup \{\bot\})$ *où* $EE = \bigcup_{\langle Init, I, l_0, I, E, U \rangle \in G} E$,

de manière à ce que pour tous $e, e' \in EE$ *avec* $Ch(e) = \langle lab_c, \langle v_1, \ldots, v_n \rangle, \langle e_1, \ldots, e_m \rangle \rangle$ *et* $Ch(e') = \langle lab'_c, \langle v'_1, \ldots, v'_{n'} \rangle, \langle e'_1, \ldots, e'_{m'} \rangle \rangle$*, si* lab_c *et* lab'_c *sont complémentaires alors* $n = m'$ *et* $m = n'$.

Un système d'XTG est donc donné par un état global décrit par *GInit*, un ensemble G d'XTG individuels et une fonction Ch associant des expressions d'échange de valeurs à des arêtes des graphes. Si $Ch(e) = \bot$ alors aucun échange de valeurs n'est associé avec e. La contrainte finale de la définition garantit que des expressions aux étiquettes complémentaires ont le même type. Nous supposons que les identificateurs des états de contrôle et des variables locales des XTG sont globalement uniques. Poursuivant les définitions sémantiques, nous définissons une fonction de synchronisation comme suit :

Définition 1.8 (synchronisation)**.** *Soient* $vp1 = \langle lab_c, \langle v_1, ..., v_n \rangle, \langle e_1, ..., e_m \rangle \rangle \in VP_V$ *et* $vp1 = \langle lab'_c, \langle v'_1, ..., v'_{n'} \rangle, \langle e'_1, ..., e'_{m'} \rangle \rangle \in VP_{V'}$*. Par* $sync(vp1, vp2) \in (2^{(V \cup V') \times Expr} \cup \{\bot\})$ *nous dénotons*

$$sync(vp1, vp2) = \begin{cases} \bigcup_{i \in \{1,\ldots,n\}} \langle v_i, e'_i \rangle \cup \bigcup_{j \in \{1,\ldots,m\}} \langle v'_j, e_j \rangle & \text{si } lab_c \text{ et } lab'_c \text{ sont complémentaires} \\ \bot & \text{sinon} \end{cases}$$

$sync(vp1, vp2)$ égale $\bot$ si $vp1$ et $vp2$ ne communiquent pas, c'est à dire si les étiquettes de synchronisation ne sont pas complémentaires. Si les expressions communiquent alors une mise à jour est produite qui résulte de la combinaison

des deux expressions. On notera que dans ce cas la définition 1.7 garantit que $n = m'$ et $m = n'$.

Définition 1.9 (sémantique de la composition parallèle)**.** *Soit un système étendu d'XTG* $\mathcal{X} = \langle GInit, \{gr_1, ..., gr_n\}, Ch\rangle$ *où* $gr_i = \langle Init_i, L_i, l_{i0}, I_i, E_i, U_i\rangle$*. Le graphe global pour* $\mathcal{X}$ *est aussi un XTG* $\langle Init, L, l_0, I, E, U\rangle$ *avec*

- $Init = \bigwedge_{i=1}^{n} Init_i$
- $L = \prod_{i=1}^{n} L_i$
- $l_0 = \langle l_{10}, ..., l_{n0}\rangle$
- $I(\langle l_1, ..., l_n\rangle) = \bigwedge_{i=1}^{n} I_i(l_i)$ *pour tout* $\langle l_1, ..., l_n\rangle \in L$
- *les ensembles* E *et* U *sont définis comme suit :*
  - *Pour toute arête* $e_i = \langle l_i, g, u, l_i'\rangle \in E_i$ *avec* $Ch(e) = \perp$*, le graphe global contient des arêtes* $e = \langle\langle l_1, ..., l_n\rangle, g, u, \langle l_1', ..., l_n'\rangle\rangle$ *pour tous* $l_j \in L_j$ *(*$j \in \{1, ..., n\} \setminus \{i\}$*), et* $l_j' = l_j$ *pour tous ces* j*. Aussi,* $e \in U$ *ssi* $e_i \in U_i$ *pour toutes ces arêtes* e*.*
  - *S'il existe deux arêtes* $e_i = \langle l_i, g_i, u_i, l_i'\rangle \in E_i$ *et* $e_j = \langle l_j, g_j, u_j, l_j'\rangle \in E_j$ *pour* $i, j \in \{1, ..., n\}$*,* $i \neq j$ *avec* $sync(Ch(e_1), Ch(e_2)) \neq \perp$ *alors* E *contient des arêtes* $e = \langle\langle l_1, ..., l_n\rangle, g, u, \langle l_1', ..., l_n'\rangle\rangle$ *où* $g = g_1 \wedge g_2$ *et* $u = u_1 \cup u_2 \cup sync(Ch(e_1), Ch(e_2))$ *pour tous* $l_k \in L_k$ *(*$k \in \{1, ..., n\} \setminus \{i, j\}$*) et* $l_k' = l_k$ *pour tous ces* k*. Aussi,* $e \in U$ *ssi* $e_i \in U_i$ *ou* $e_j \in U_j$*, pour toutes ces arêtes* e*.*

Les définitions des ensembles E et U méritent quelques explications. Une arête du graphe global correspond soit à une arête d'un des graphes d'origine, soit – comme conséquence d'une synchronisation – de deux arêtes correspondantes de deux graphes différents. Dans le premier cas, l'arête originale n'est pas associée avec une expression de communication, car les arêtes de ce dernier type doivent être synchronisées. L'arête globale résultante comporte alors la garde, la mise à jour et l'attribut d'urgence de l'arête originale. Dans le cas d'une arête correspondante à la communication de deux arêtes, la garde de l'arête globale et la conjonction des deux gardes originales, et la mise à jour combine les mises à jour originales avec la mise à jour résultant de la synchronisation. L'arête globale est urgente si et seulement si l'une des deux arêtes originales l'est.

1.2.2 PDT : représentation abstraite des XTG

Les techniques habituelles de *model checking* ne s'appliquent pas aux XTG à cause de leur modèle riche de données. Nous allons maintenant introduire les PDT (*predicate diagrams for timed systems*, diagrammes de prédicats pour les systèmes temporisés) qui représenteront les abstractions booléennes des XTG. La

vérification se réduit alors à (a) établir la correction de la représentation abstraite et (b) évaluer la propriété d'intérêt sur cette représentation abstraite. Nous utiliserons les techniques d'abstraction booléenne et des techniques mécanisées de la preuve pour résoudre le sous-problème (a). Dans cette section nous allons identifier un ensemble de conditions non-temporelles et suffisantes. Le sous-problème (b) sera résolu en utilisant la technique de *model checking* car nos abstractions donnent lieu à des modèles finis.

La définition formelle des PDT est générique par rapport à un ensemble L qui représente les états (ou plus précisément les n-uplets d'états) de contrôle de l'XTG, ainsi que par rapport à un ensemble $\mathcal{P}$ de prédicats (expressions booléennes). Par $\overline{\mathcal{P}}$ on dénotera l'ensemble formé par les prédicats dans $\mathcal{P}$ et leurs négations.

Définition 1.10. *Soient L et $\mathcal{P}$ deux ensembles finis. Un* PDT *(pour L et $\mathcal{P}$) est un quadruplet $\langle N, N_0, R_\mu, R_\tau \rangle$ où :*

- *$N \subseteq L \times 2^{\overline{\mathcal{P}}}$ est un ensemble fini de nœuds, chaque nœud étant un couple $\langle l, P \rangle$ où $l \in L$ et $P \subseteq \overline{\mathcal{P}}$,*
- *$N_0 \subseteq N$ est l'ensemble des nœuds initiaux,*
- *$R_\mu, R_\tau \subseteq N \times N$ sont deux relations décrivant les transitions discrètes et de passage de temps du PDT. La relation R_τ doit être réflexive. On écrira $n \rightarrow_\mu n'$ et $n \rightarrow_\tau n'$ pour $(n, n') \in R_\mu$ et $(n, n') \in R_\tau$.*

Une exécution *d'un PDT est une suite infinie*

$$\sigma \quad = \quad n_0 \xrightarrow{lab_0} n_1 \xrightarrow{lab_1} n_2 \ldots$$

où $n_0 \in N_0$, $lab_i \in \{\mu, \tau\}$ et $n_i \rightarrow_{lab_i} n_{i+1}$ pour tout $i \in \mathbb{N}$.

Ainsi, un PDT est un système de transitions étiqueté avec deux relations de transition. Un nœud d'un PDT représente un ensemble d'états de l'XTG en indiquant les états actifs de contrôle et quelques prédicats vérifiés par ces états. Les relations de transition correspondent aux transitions discrètes et de passage de temps de l'XTG. A chaque nœud on associe une boucle τ implicite.

Conformité : lien entre XTG et PDT

Un PDT décrit une représentation abstraite d'un XTG. Afin de démontrer à l'aide de ce PDT – pour une spécification (un XTG) et une formule de la logique temporelle données – que la formule est vraie pour la spécification, le PDT doit vérifier deux conditions : d'abord, toutes les exécutions de l'XTG doivent être représentées dans le PDT. Puis, toute exécution du PDT doit vérifier la formule d'intérêt.

Pour établir cette deuxième condition nous considérons toutes les étiquettes du PDT comme étant des variables booléennes. Ceci nous permettra de coder le PDT sous la forme d'un système de transitions fini dont les propriétés temporelles peuvent être établies par *model checking*.

Nous allons à présent définir la conformité d'un PDT par rapport à un XTG ; cette relation représente la correction de l'abstraction. Nous allons aussi établir un ensemble de conditions de vérification garantissant la conformité. L'objectif de cette définition est de garantir que toute propriété vérifiée par le PDT est aussi vraie pour l'XTG. Puisque nous nous intéressons à la vérification de propriétés écrites en logique temporelle linéaire et de telles propriétés sont vérifiées par un XTG si elles sont vérifiées par chacune des exécutions de l'XTG, nous allons vérifier que toute exécution de l'XTG est représentée dans le PDT. La définition suivante formalise cette intuition.

Définition 1.11. *Soient $\mathcal{X}$ un XTG, Δ un PDT et $\pi = \langle l_0, \rho_0\rangle \xrightarrow{\lambda_0} \langle l_1, \rho_1\rangle \dots$ une exécution de $\mathcal{X}$. Une exécution $\sigma = n_0 \xrightarrow{lab_0} n_1 \dots$ de Δ est une* trace *de π si*

- *$n_i = \langle l_i, P_i\rangle$ pour un certain $P_i \subseteq \overline{\mathcal{P}}$ tel que $[\![p]\!]_{\rho_i} = tt$ pour tous $p \in P_i$ et $i \in \mathbb{N}$, c'est à dire que les états de π et les nœuds de σ sont d'accord sur les états de contrôle et que les prédicats des n_i sont vérifiées dans les environnements correspondants de π, et*
- *$lab_i = \mu$ si $\lambda_i = \mu$ et $lab_i = \tau$ si $\lambda_i \in \mathbb{R}^{\geq 0}$, c'est à dire que les deux exécutions sont d'accord sur quelles des transitions sont discrètes et quelles transitions correspondent à des laps de temps.*

Δ est conforme à $\mathcal{X}$ si pour toute exécution de $\mathcal{X}$ il existe une trace dans Δ.

Afin d'établir la conformité d'après la définition 1.11 il faut examiner toutes les exécutions de l'XTG. En pratique il est intéressant de trouver un ensemble de conditions de vérification suffisantes pour garantir la conformité. Le théorème suivant donne de telles conditions. L'idée est de vérifier que tout état initial possible de l'XTG est représenté par un nœud initial du PDT. De manière inductive, étant donné un état s de l'XTG représenté par un nœud n du PDT et une transition entre s et un état s' de l'XTG, cette transition doit être représentée par une transition du PDT émanant du nœud n. Dans la formalisation des conditions de vérification nous nous servons de deux copies V' et V'' de l'ensemble V des variables dont les éléments sont décorés par des primes simples et doubles (v' et v'' pour tout $v \in V$). Pour un ensemble P de prédicats nous dénotons également par P la conjonction de ces prédicats, et écrivons P' et P'' la formule obtenue en remplaçant chaque variable $v \in V$ par sa copie v' ou v''.

Théorème 1.12 (conformité)**.** *Soit $\mathcal{X}$ un XTG comportant m processus $P_i = \langle Init_i, L_i, l_{0,i}, I_i, E_i, U_i\rangle$. Le PDT $\Delta = \langle N, N_0, R_\mu, R_\tau\rangle$ sur $L_1 \times \dots \times L_m$ et un ensemble $\mathcal{P}$ de prédicats est conforme à $\mathcal{X}$ si toutes les conditions suivantes sont vraies :*

1. $\displaystyle\bigwedge_{j=1}^{m} Init_j \wedge I(l_{0,j}) \;\Rightarrow \bigvee_{\langle l_{0,1},\dots,l_{0,m},P\rangle \in N_0} P$

   La conjonction des conditions initiales de $\mathcal{X}$ et des invariants associés aux états initiaux implique que les prédicats d'un des nœuds initiaux de Δ correspondant aux états initiaux de $\mathcal{X}$ sont vérifiées.

2. *Pour tout nœud $n = \langle l_1, \ldots, l_m, P\rangle$ de Δ et toute arête $\langle l_i, g, u, l_i'\rangle$ d'un processus P_i, soit V_u l'ensemble des variables v mises à jour par u (c'est à dire telles que $\langle v, e\rangle \in u$ pour un certain e), et soit N' l'ensemble des nœuds $n' = \langle l_1', \ldots, l_m', Q\rangle$ avec $l_j' = l_j$ pour $j \neq i$ et $n \rightarrow_\mu n'$.*

$$P \wedge g \wedge \bigwedge_{j=1}^{m} (I(l_j) \wedge I'(l_j')) \wedge \bigwedge_{\langle v,e\rangle \in u} v' = e \wedge \bigwedge_{v \in V \setminus V_u} v' = v \;\Rightarrow \bigvee_{\langle l_1', \ldots, l_m', Q\rangle \in N'} Q'$$

Les prédicats du nœud n, les invariants associés aux états actifs avant et après la transition de $\mathcal{X}$, ainsi que la mise à jour engendrée par celle-ci impliquent les prédicats d'un certain nœud de N'.

3. *Pour tout nœud $n = \langle l_1, \ldots, l_m, P\rangle$ de Δ soit N'' l'ensemble des nœuds $n'' = \langle l_1, \ldots, l_m, Q\rangle$ associés aux mêmes états de contrôle que n et tels que $n \rightarrow_\tau n''$.*

$$\begin{aligned} & P \wedge \delta \in \mathbb{R}^{\geq 0} \wedge \bigwedge_{c \in V_c} c' = c + \delta \wedge \bigwedge_{v \in V \setminus V_c} v' = v \wedge \bigwedge_{j=1}^{m} I(l_j) \wedge I'(l_j) \\ & \wedge\ \forall \varepsilon \leq \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow \bigwedge_{j=1}^{m} I''(l_j) \\ & \wedge\ \forall \varepsilon < \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow \bigwedge_{j=1}^{m} \bigwedge_{\langle l_j, g, u, l_j'\rangle \in U_j} \neg g'' \\ \Rightarrow \quad & \bigvee_{\langle l_1, \ldots, l_m, Q\rangle \in N''} Q' \end{aligned}$$

Pour toute transition correspondant à un laps de temps par δ telle qu'aucune arête urgente de $\mathcal{X}$ émanant des états actifs dans n n'est activée avant, les prédicats de n et les invariants des états actifs dans n (pour toutes les valeurs de temps intermédiaires) impliquent que les prédicats d'un certain nœud dans N'' sont vérifiés.

Raffinement structurel

Au-delà de leur utilité dans la vérification de propriétés d'XTG à travers un PDT conforme, les PDT peuvent aussi être utiles afin de comparer deux modèles d'un même système écrits à deux niveaux différents d'abstraction.

On écrira $\Delta^1 \preceq \Delta^2$ ("Δ^1 raffine Δ^2") si toute exécution du PDT Δ^1 est aussi une exécution de Δ^2. On suppose que l'ensemble des prédicats sous-jacent à Δ^1 étend celui pour Δ^2. Comme pour la preuve de la conformité, nous nous intéressons à des conditions "locales" afin d'établir le raffinement entre deux diagrammes, et nous définissons la notion de raffinement structurel comme suit.

Définition 1.13 (raffinement structurel)**.** *Soient $\Delta^i = \langle {N^i}, {N_0}^i, {R_\mu}^i, {R_\tau}^i\rangle$ (pour $i = 1, 2$) deux PDT sur L et $\mathcal{P}$, et soit $f : N^1 \rightarrow N^2$ une fonction. Le PDT Δ^1*

est un raffinement structurel de Δ^2 *à travers* f *si les conditions suivantes sont vérifiées :*

1. $\models n \Rightarrow f(n)$ *pour tout nœud* $n \in N^1$,
2. $f(n) \in N_0^2$ *pour tout* $n \in N_0^1$,
3. *pour tous* $n, n' \in N^1$ *avec* $(n, n') \in R_\varrho^1$ *(où* $\varrho \in \{\mu, \tau\}$*) on a* $(f(n), f(n')) \in (R_{\mu\cup\tau}^2)^=$, *la clôture réflexive de l'union des relations de transition dans* Δ^2.

Nous dirons que Δ^1 *est un raffinement structurel de* Δ^2 *s'il est un raffinement structurel de* Δ^2 *à travers une certaine fonction* $f : N^1 \to N^2$.

Les conditions 1–3 assurent qu'il existe une correspondance forte entre les graphes de transitions des deux diagrammes : les étiquettes des nœuds de Δ^1 impliquent ceux des nœuds correspondants de Δ^2, les nœuds initiaux de Δ^1 correspondent aux nœuds initiaux de Δ^2 et les transitions de Δ^1 se retrouvent dans Δ^2, au bégaiement près.

Le théorème suivant formalise cette observation. En particulier, toutes les propriétés temporelles exprimées en un langage de logique temporelle non sensible au bégaiement et qui ne font référence qu'aux prédicats inclus dans les étiquettes des nœuds du diagramme Δ^2 sont préservées dans Δ^1.

Théorème 1.14. *Si* Δ^1 *est un raffinement structurel de* Δ^2 *à travers la fonction* f *alors pour toute exécution* $n_0 \xrightarrow{lab_0} n_1 \xrightarrow{lab_1} \ldots$ *de* Δ_1 *la suite* $(f(n_0), f(n_1), \ldots)$ *est un chemin dans* Δ^2 *qui peut contenir des répétitions de nœuds.*

1.2.3 *Model checking* : évaluation de propriétés

Un PDT Δ conforme à un XTG $\mathcal{X}$ peut servir de base à l'évaluation de propriétés de correction de $\mathcal{X}$. Nous supposons que les propriétés qui nous intéressent sont exprimées en logique temporelle linéaire LTL (éventuellement dans un fragment clos sous bégaiement, par exemple sans l'utilisation de la modalité «next») et qu'elles sont exprimées avec des prédicats dans $\mathcal{P}$. Ceci nous permet de considérer les prédicats dans les nœuds de Δ comme étant des propositions atomiques non-interprétées.

Considérant Δ comme un système fini de transitions les propriétés temporelles de ses exécutions peuvent être établies en utilisant des techniques de *model checking* pour LTL. Aussi, Δ peut être codé dans le langage de modélisation d'outils standard de *model checking*. Pour nos expériences nous utilisons soit Spin à travers l'outil DIXIT [27] ou le *model checker* PMC [12].

Logique temporelle linéaire

La logique temporelle linéaire LTL [29] est très populaire pour exprimer des assertions sur les comportements temporels de systèmes. A partir d'un ensemble fini AP de propositions atomiques, les formules de LTL sont définies de manière inductive comme suit :

- Chaque élément de l'ensemble AP est une formule.
- Si φ et ψ sont des formules alors $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\bigcirc\varphi$ et φ U ψ le sont aussi.

Une *interprétation* pour une formule LTL est un ω-mot $\zeta = x_0, x_1, \ldots$ sur l'alphabet 2^{AP}, c'est à dire une fonction des entiers vers 2^{AP}. On dénotera par ζ^i le suffixe de ζ à partir de x_i. La sémantique des formules LTL est définie comme suit :

- $\zeta \models A$ ssi $A \in x_0$, pour tout $A \in AP$,
- $\zeta \models \neg\varphi$ ssi $\zeta \not\models \varphi$,
- $\zeta \models \varphi \wedge \psi$ ssi $\zeta \models \varphi$ et $\zeta \models \psi$,
- $\zeta \models \varphi \vee \psi$ ssi $\zeta \models \varphi$ ou $\zeta \models \psi$,
- $\zeta \models \bigcirc\varphi$ ssi $\zeta^1 \models \varphi$,
- $\zeta \models \varphi$ U ψ ssi il existe un certain $i \geq 0$ avec $\zeta^i \models \psi$ et $\zeta^j \models \varphi$ pour tout $0 \leq j < i$.

Nous écrivons F pour $A \wedge \neg A$ (pour une proposition atomique A quelconque) et T pour $\neg\mathsf{F}$. Aussi, $\Diamond\varphi$ est une abréviation pour T U φ, $\Box\varphi$ est défini comme $\neg\Diamond\neg\varphi$ et $\varphi \Rightarrow \psi$ comme $\neg\varphi \vee \psi$.

La correction des approches de vérification fondées sur l'abstraction repose généralement sur la propriété suivante : quand le système abstrait vérifie une formule alors le système concret la vérifie aussi. Comme les formules LTL sont interprétées sur toutes les exécutions possibles, il suffit de démontrer que toute exécution du système concret correspond à une certaine exécution du système abstrait. C'est la propriété de correction que nous avons introduite pour les PDT, et nous pouvons par conséquence affirmer que si Δ vérifie une certaine propriété φ alors $\mathcal{X}$ la vérifie aussi.

Cependant, si Δ ne vérifie pas une certaine propriété, le *model checker* fournira un contre-exemple et nous essayons de trouver une exécution concrète correspondante à cette trace. En fait, le contre-exemple abstrait ne correspondra pas forcément à un contre-exemple concret car certains détails ne sont pas représentés dans l'abstraction. Néanmoins, l'analyse du contre-exemple abstrait peut aider à affiner l'abstraction, et de telles méthodes sont bien connues et souvent utilisées dans d'autres travaux [28, 15, 11, 19].

Dans ce qui suit nous allons proposer une méthodologie (semi-automatique) différente et nouvelle dont l'objectif est d'éviter de faux résultats négatifs. L'idée générale est d'insister sur la préservation de la propriété φ lors de la génération de la représentation abstraite. Le chapitre suivant expliquera les détails de cette approche.

1.3 Raffinement abstrait itératif

Nous allons proposer une méthodologie (semi-automatique) nouvelle et utile en pratique afin de construire une abstraction complète qui nous servira pour la vérification de propriétés de sûreté et de vivacité du système concret. Notre propo-

sition se concrétise par une approche de la vérification intégrant des techniques d'abstraction booléenne, de preuve déductive et de *model checking* qui permet de vérifier une classe riche de propriétés de sûreté et de vivacité de systèmes temporisés. Son objectif est de pouvoir calculer une abstraction finie dont la conformité avec le système concret est garantie par raffinements successifs. Nous appelons cette méthode le *raffinement abstrait itératif* (iterative abstract refinement, IAR).

Notre méthode est fondée sur une technique simple, efficace et précise pour la génération d'abstractions qui préserve les propriétés et qui garantit la correction de l'abstraction. L'idée est de partir d'une sur-approximation initiale (que nous appelons souvent abstraction réduite dans cette thèse) du modèle et puis de vérifier deux propos : (1) la conformité entre la représentation abstraite et le modèle concret et (2) l'évaluation des propriétés requises sur le modèle abstrait. Si la représentation abstraite est conforme au modèle concret et si les propriétés qui nous intéressent peuvent être vérifiées sur le modèle abstrait alors elles sont aussi vraies pour le système concret et nous pouvons dire qu'un tel modèle abstrait est la représentation abstraite complète.

Soit $\mathcal{X}$ une spécification d'un système temporisé sous forme d'XTG et $\mathcal{F}$ une propriété. Rappelons que dans une perspective de *model checking* la preuve que $\mathcal{X}$ vérifie $\mathcal{F}$ peut être considérée comme démontrant la validité de $\mathcal{X} \models \mathcal{F}$. L'approche IAR établit cette validité à l'aide d'un PDT Δ en démontrant à la fois que Δ est une abstraction correcte (c'est à dire que nous allons trouver un Δ tel que toute exécution de $\mathcal{X}$ correspond à une exécution de Δ) et en vérifiant que toute exécution de Δ est un modèle de $\mathcal{F}$.

Pour la preuve de la correction, nous considérons les étiquettes des nœuds de Δ comme des prédicats d'une abstraction booléenne sur l'espace d'états de $\mathcal{X}$, pouvant ramener l'inclusion des traces à un ensemble d'obligations de preuve en un langage de premier ordre sur des états et des transitions individuels. Ainsi, cette étape utilise des techniques déductives. Pour l'autre étape, nous considérons les étiquettes des nœuds comme des variables booléennes et Δ comme un système fini de transitions. La propriété $\mathcal{F}$ du système est alors établie par *model checking* du PDT Δ. Une des contributions de cette thèse est de montrer que le format de représentation choisi est utile pour démontrer non seulement des propriétés de sûreté mais aussi de vivacité. Il n'est guère surprenant que ces dernières sont beaucoup plus difficiles à établir que les propriétés de sûreté. Nous allons expliquer ce que sont les conditions auxiliaires et définir plusieurs concepts nouveaux que nous utilisons afin de démontrer des propriétés de vivacité à l'aide de Δ.

La contribution centrale de IAR est de pouvoir construire une représentation abstraite garantissant que les propriétés LTL déduites du modèle abstrait sont aussi vraies pour le modèle concret. Avant de parler plus en détail de IAR nous illustrons trois modèles différents d'un système par la figure 1.2 qui contient une abstraction réduite, une abstraction complète et le modèle concret d'un système.

Le modèle abstrait illustré par la figure 1.2.(a) résulte d'une approximation ; par conséquent, la préservation des exécutions du modèle concret (c) par le modèle (a) n'est pas garantie. Le modèle de la figure 1.2.(b) est obtenu à partir du modèle

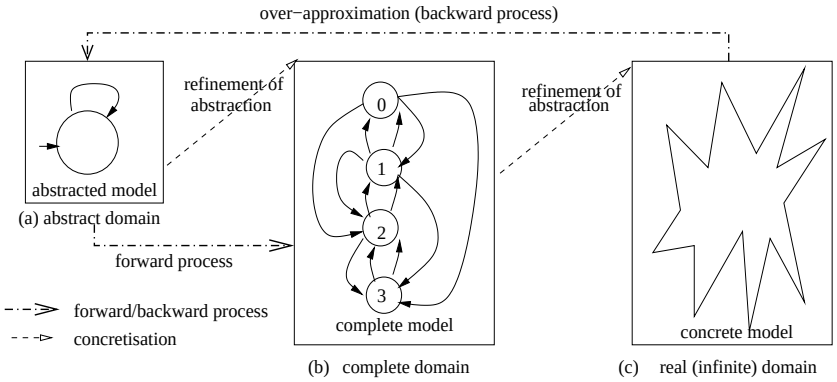


Fig. 1.2: *Modèles abstrait, complet et concret*

abstrait (a) par des raffinements successifs jusqu'à ce qu'il corresponde au modèle concret (c) et que les propriétés puissent être vérifiées. Nous nous attendons à pouvoir obtenir un tel modèle complet (b) en utilisant la méthode IAR.

La figure 1.3 (qui est liée à la figure 1.2) illustre le cadre général de la méthode IAR. Elle prend trois paramètres d'entrée : la spécification du système concret sous la forme d'un XTG $\mathcal{X}$, les propriétés à vérifier décrivant le comportement attendu et un ensemble fini de prédicats d'abstraction.

La méthode IAR consiste en l'application de deux processus (dits en avant et en arrière) : une première sur-approximation (la représentation abstraite réduite) peut être obtenue à la main à partir de $\mathcal{X}$ à l'aide d'un processus en arrière. Partant de cette abstraction réduite, IAR itère le processus en avant jusqu'à ce que la représentation abstraite devienne complète :

Processus en arrière. Une abstraction triviale et incomplète de $\mathcal{X}$ est obtenue par abstraction booléenne, voir la direction de (c) vers (a) de la figure 1.2. Il en résulte un PDT Δ comme défini dans le chapitre 1.2.2 ; la figure 5.1.(a) de la section 5.4 illustre comment obtenir une telle abstraction à travers un petit exemple. Cependant, ce PDT Δ réduit obtenu par le processus en arrière ne garantit pas la condition de complétude car tous les chemins de $\mathcal{X}$ ne sont pas représentés. Un autre processus, dit en avant, est nécessaire afin d'affiner de telles représentations incomplètes.

Processus en avant. Pour un PDT Δ obtenu, la méthode IAR vérifie s'il est complet, c'est à dire que les propriétés sont décidables et préservées par Δ et que Δ est conforme à $\mathcal{X}$. Dans ce cas, une représentation abstraite correcte a été obtenue et la vérification réussit. Si Δ n'est pas complet, elle assure la complétude (affinant Δ) au travers les deux opérations de *scindage* et d'*exclusion*.

Pendant que la méthode IAR cherche à assurer qu'une représentation abstraite

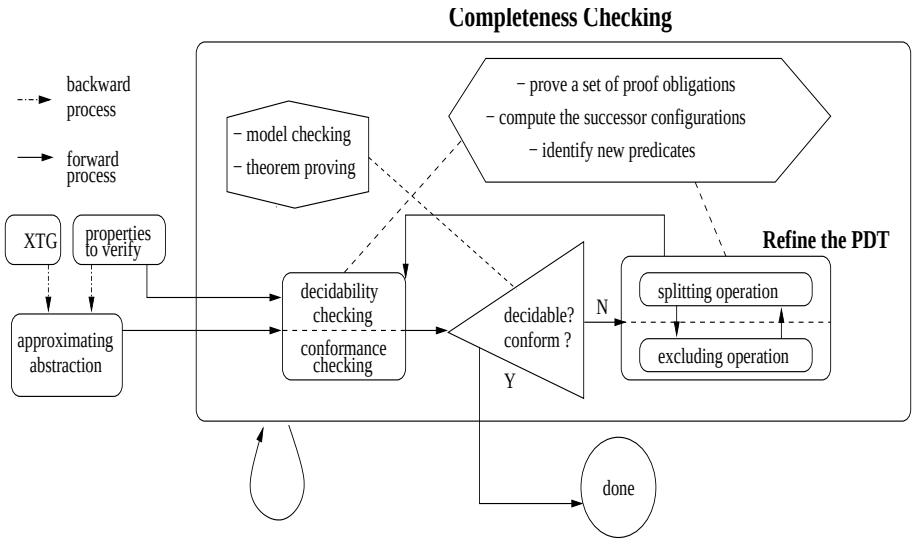


Fig. 1.3: *IAR : présentation générale*

par un PDT est complète, elle vérifie (1) le problème de *décidabilité* sur Δ et (2) la conformité entre Δ et $\mathcal{X}$.

- Si IAR n'arrive pas à démontrer ces deux problèmes, alors le PDT est scindé par rapport à des prédicats d'abstraction afin d'enrichir Δ en ajoutant des détails.
- Pendant cette opération, certaines obligations de preuve (des conditions de vérification formulées en logique de premier ordre) sont démontrées afin d'exclure des transitions dupliquées pendant le scindage. Les deux opérations de scindage et d'exclusion sont itérées jusqu'à ce que le PDT devienne complet.

La méthode IAR termine lorsqu'elle obtient un PDT Δ dont les chemins correspondent à ceux de $\mathcal{X}$ de manière à ce que les propriétés vérifiées par Δ le soient aussi par $\mathcal{X}$. Plus spécifiquement, partant de la première abstraction, IAR génère des abstractions en ajoutant de nouveaux nœuds et arêtes correspondant à $\mathcal{X}$. Ceci continue jusqu'à ce que les propriétés soient vérifiées par rapport à Δ et qu'aucun nœud et aucune arête ne doivent être ajoutés.

La méthode IAR construit donc une représentation abstraite à partir du modèle concret et justifie sa complétude en utilisant une combinaison de procédures de décision et la vérification de la conformité. Dans la suite de cette thèse nous allons étudier à travers quelques exemples comment la méthode IAR établit un cadre pour la vérification.

Dans ce qui suit, le raffinement d'abstractions guidé par les contre-exemples (counterexample-guided abstraction refinement, CEGAR) n'est pas vraiment né-

cessaire car nous utilisons la propriété de sûreté (décrivant les comportements attendus) comme l'un des prédicats pour la construction du modèle abstrait. Ceci suffit à garantir que le PDT représente toutes les exécutions de l'XTG, y compris le comportement attendu. Cependant nous pensons que des améliorations notables pourraient être obtenues en utilisant CEGAR car cela résulterait en de meilleurs raffinements.

1.4 Application d'IAR

Nous illustrons l'application de la méthode IAR à travers le protocole temporisé d'exclusion mutuelle bien connu qui est dû à Fischer [8, 42] en vérifiant des propriétés de sûreté et de vivacité.

1.4.1 Le protocole d'exclusion mutuelle de Fischer

Nous considérons d'abord la propriété de sûreté pour une version simplifiée du protocole ne comportant que deux processus. La figure 1.4 illustre la structure du processus i (où $i = 1, 2$) avec une variable k partagée par les deux processus, alors que c_i est une horloge locale du processus i. Les contraintes temporelles garantissent que tous les processus à l'état 2 (l_{i2}) attendent jusqu'à ce que tous les processus à l'état 1 (l_{i1}) aient pris la transition à l'état 2.

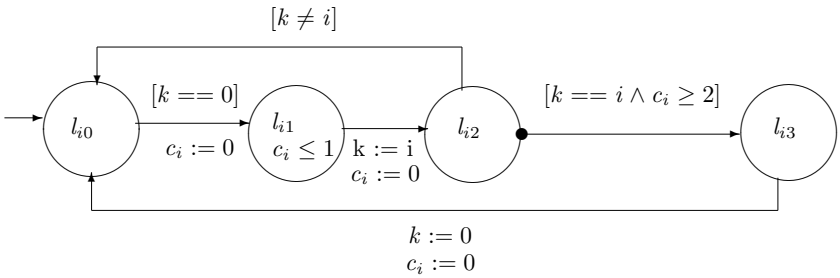
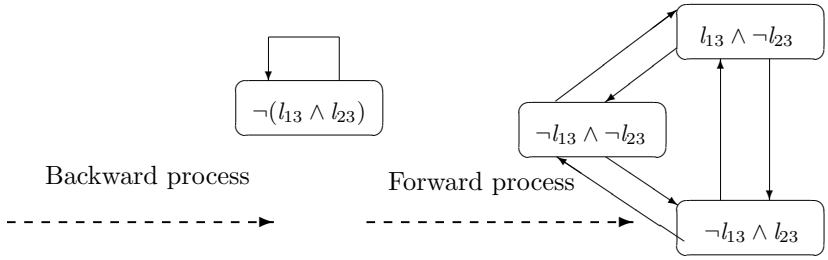


Fig. 1.4: *La spécification XTG du protocole de Fischer*

L'idée du protocole est la suivante : pendant une première phase chacun des processus essaie d'enregistrer son identifiant dans la variable partagée k. Dans une deuxième phase chaque processus vérifie si son identifiant figure toujours dans k après un certain laps de temps, puis entre en section critique. L'objectif du protocole est de garantir sa sûreté de manière à ce que jamais plus d'un processus ne se trouve en section critique, ce qui est traduit par la formule LTL

$$\Box\neg(\mathbf{at}\, l_{13} \wedge \mathbf{at}\, l_{23}). \tag{1.1}$$

Afin d'utiliser la méthode IAR nous spécifions d'abord le protocole dans l'XTG $\mathcal{X}$ montré dans la figure 1.4. Partant de $\mathcal{X}$ nous obtenons le premier PDT Δ



(a) The first PDT of XTG (b) The forward-refinement of (a)

Fig. 1.5: *Les processus en arrière et en avant pour le protocole de Fischer*

(au travers une abstraction en arrière) représentant la formule LTL (1.1) qui décrit l'exclusion mutuelle entre les sections critiques des deux processus. Ce Δ est illustré dans la figure 1.5.(a) ; il garantit clairement la propriété.

Le processus en avant est entamé sur le PDT de la figure 1.5.(a) afin de l'enrichir jusqu'à ce qu'il devienne complet. Pour la construction d'un PDT complet nous considérons les deux aspects de la décidabilité de la propriété de correction sur le PDT et la conformité entre le PDT et l'XTG $\mathcal{X}$. En effet nous voyons que le premier aspect est déjà vérifié pour le PDT Δ de la figure 1.5.(a) obtenu par le processus en arrière.

Nous considérons maintenant le deuxième aspect, la correction de la représentation. La première application du processus en avant au PDT de la figure 1.5.(a) résulte en le PDT montré dans la figure 1.5.(b). Bien que le simple diagramme de la figure 1.5.(a) vérifie la propriété de haut niveau, le degré de conformité par rapport au système concret est très bas.

Il nous faut donc enrichir ce diagramme (a) afin d'améliorer sa correspondance par rapport à $\mathcal{X}$. La première étape de raffinement est représentée dans le diagramme montré dans la figure 1.5.(b) qui élimine la possibilité qu'un processus donne la main immédiatement à l'autre. Nous ne montrons plus les boucles sur chacun des nœuds car elles sont implicites dans la définition d'un PDT. Il est facile de voir que le diagramme (b) est un raffinement structurel de (a) car les prédicats de chaque étiquette impliquent $\neg(l_{13} \wedge l_{23})$ et chaque transition est permise par la boucle sur l'unique nœud du diagramme initial.

Or, ce diagramme (b) n'est toujours pas assez détaillé (précis) et ne satisfait pas non plus de propriété de vivacité. Nous envisageons alors quelques événements et invariants (prédicats) qui seraient à ajouter au diagramme (b) afin de l'enrichir en vue de la construction d'un PDT complet, conforme à $\mathcal{X}$ et vérifiant les propriétés de sûreté et de vivacité. Les nœuds à ajouter seraient suggérés en essayant de démontrer les conditions de conformité énoncées dans le théorème 1.12.

Afin d'obtenir un diagramme pleinement affiné nous continuons à appliquer

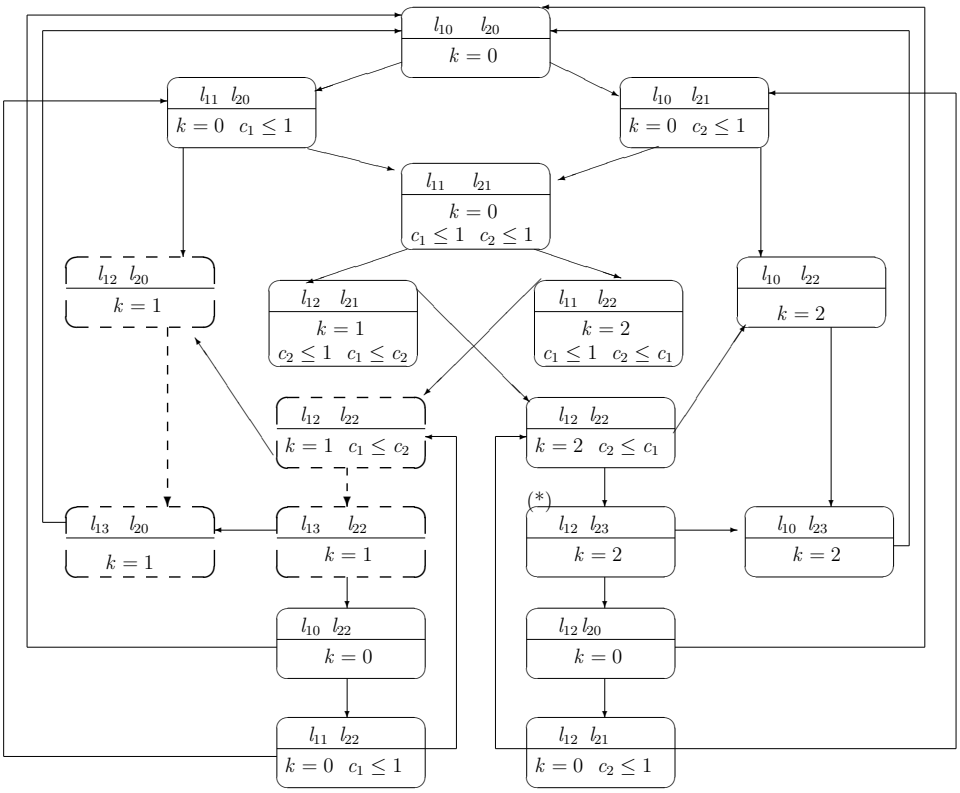


Fig. 1.6: *Diagramme Δ_1 pour le protocole de Fischer (voir Fig. 1.4).*

la même procédure, explorant les nœuds du diagramme actuel et ajoutant des événements et invariants indiquant quel processus a demandé l'accès à la section critique et quel processus est gagnant, en analysant les exécutions de $\mathcal{X}$. Enfin, nous arrivons au diagramme Δ_1 de la figure 1.6. (A noter que les chapitres 5 et 6 de l'annexe technique expliquent davantage de détails sur la génération de diagrammes.)

Pour la vérification de la conformité, l'ensemble des conditions de vérification énoncées dans le théorème 1.12 est vérifié à l'aide de l'outil automatique CVC-LITE ; la preuve entière apparaît dans l'annexe A. (Il faut noter que toutes les arêtes correspondant au passage de temps du diagramme Δ_1 de la figure 1.6 sont des boucles sur les nœuds que nous avons convenu de ne pas montrer de manière explicite.)

La méthode peut être outillée de deux façons : la première méthode consiste en l'application de l'outil DIXIT en utilisant les *model checker* Spin ou PMC. Dans notre exemple DIXIT confirme que le diagramme vérifie l'exclusion mutuelle. La

deuxième méthode est d'utiliser CVC-LITE encore une fois afin de déduire la propriété qui nous intéresse à partir des étiquettes des nœuds. Par exemple, afin de vérifier la validité de la formule (1.1) nous demandons à CVC-LITE de vérifier la formule

$$\neg l_{13} \vee \neg l_{23} \tag{1.2}$$

sur chacun des nœuds du diagramme Δ_1. Comme chacun des nœuds vérifie cette formule, nous concluons que le protocole de Fischer garantit l'exclusion mutuelle.

Conditions auxiliaires pour la vérification de propriétés de vivacité

Nous considérons à présent la vérification des propriétés de vivacité sur la base de PDT. La formulation précise de la vivacité impose de se focaliser sur un processus donné et de demander qu'une requête d'un processus sera honorée un jour.

La vérification d'une telle propriété de vivacité nécessite des considérations supplémentaires. En effet, les spécifications XTG assurent la vivacité en utilisant des contraintes d'horloges, ainsi que des conditions d'urgence (marquées par «asap»), associées aux états et transitions. Rappelons que chaque nœud d'un PDT porte une boucle τ implicite qui permet un bégaiement infini. De ce fait, le codage d'un PDT décrit précédemment ne permet pas immédiatement de vérifier les propriétés de vivacité.

Afin de résoudre ce problème nous introduisons une condition auxiliaire aux PDT qui repose sur des prédicats associés aux horloges. Considérons un nœud qui impose une borne supérieure à un horloge. Cette borne implique qu'aucune exécution passant par ce nœud ne peut rester à ce nœud indéfiniment. Nous appelons un tel nœud un *nœud bornant le temps*.

Définition 1.15 (nœud bornant le temps)**.** *Soit $\Delta = \langle N, N_0, R_\mu, R_\tau \rangle$ un PDT sur L et $\mathcal{P}$. Un nœud $n \in N$ est un nœud bornant le temps si son étiquette comporte un prédicat de la forme $c \leq e$ où $c \in V_c$ est un horloge est $e \in Expr$ une expression. La contrainte $c \leq e$ est dénotée par $clk(n)$ et l'expression e par $\uplus(n)$.*

La condition de la définition 1.15 introduit une notion de progrès de temps dans l'analyse des PDT ; elle est liée aux invariants des états des XTG et aux conditions d'urgence.

Nous illustrons la vérification d'une propriété de vivacité, toujours sur le même exemple. Rappelons que dans le protocole de Fischer les processus à l'état l_2 attendent que tous les processus soient passés de l_1 à l_2. Afin d'exclure le cas où tous les processus restent bloqués à l_2, nous allons vérifier la condition de vivacité qu'un des processus à l_2 finit par accéder à l_3.

Dans l'XTG modélisant le protocole de Fischer, cette propriété est assurée par la transition urgente de l_2 à l_3 qui est représentée par le point noir dans la figure 1.4.

Afin de poursuivre notre développement vers un PDT permettant de vérifier cette condition de vivacité nous ajoutons des contraintes sur les horloges à des nœuds du diagramme Δ_1, c'est à dire que nous renforçons Δ_1 par des conditions auxiliaires (voir la définition 1.15). Après cette modification, ces nœuds deviennent des nœuds bornant le temps.

Correspondant aux transitions urgentes émanant des états l_{i2} de l'XTG $\mathcal{X}$ (voir la figure 1.4), nous ajoutons les contraintes $c \leq 2$ aux nœuds du diagramme Δ_1 correspondant à ces états de contrôle. Formellement (voir la figure 1.7), nous mettons

- $\uplus(n) = 2$ pour $n \in \{n_{L1}, n_{L3}, n_{R1}, n_{R3}\}$,
- $clk(n) = c_1 \leq \uplus(n)$ pour $n \in \{n_{L1}, n_{L3}\}$,
- $clk(n) = c_2 \leq \uplus(n)$ pour $n \in \{n_{R1}, n_{R3}\}$.

Ces ajouts de prédicats sont justifiés par les conditions du théorème 1.12. En effet, ils correspondent aux invariants associés aux états de contrôle l_{i2} qui figurent à gauche des implications associées aux transitions menant aux nœuds concernés.

Le diagramme Δ_2 de la figure 1.7 a la même structure que le diagramme Δ_1 de la figure 1.6 mais avec des étiquettes étendues. Comme nous l'avons justifié précédemment, il est aisé à démontrer que le diagramme Δ_2 est lui aussi conforme au XTG $\mathcal{X}$; il impose une contrainte évitant le bégaiement infini dans des nœuds où cela est exclu. Une preuve formelle de la conformité à l'aide de CVC-Lite est donnée dans l'annexe A.

La vérification de la propriété de vivacité à l'aide du PDT Δ_2 est possible à l'aide de l'outil Dixit. Cette propriété est exprimée en LTL comme suit :

$$\Box\left(\left(\bigvee_{i=1}^{2} l_{i2}\right) \Rightarrow \Diamond\left(\bigvee_{i=1}^{2} l_{i3}\right)\right) \tag{1.3}$$

On observera que l'outil CVC-Lite n'est pas utile pour la vérification de cette formule car elle nécessite un raisonnement sur les exécutions du modèle abstrait ce qui est le domaine d'outils de *model checking*.

Dans la discussion de l'exemple du protocole de Fischer nous n'avons pas considéré une approche CEGAR afin de prendre en compte des résultats négatifs lors de la vérification. En effet, ceci n'est pas nécessaire car nous avons utilisé la propriété de sûreté parmi les prédicats d'abstractions dans le sens que la représentation abstraite Δ construite garantit cette propriété. Par conséquent nous savons d'ores et déjà que le résultat de *model checking* sur le diagramme Δ sera positif et il n'aura pas de contre-exemple abstrait à analyser.

1.4.2 Diagrammes de prédicats pour les systèmes paramétrés

La vérification de systèmes paramétrés s'effectue souvent à la main ou en utilisant un assistant interactif de preuve [33, 47]. Des méthodes fondées sur la construction semi-automatique d'un invariant de processus sont proposées par [58]. Cependant, il n'est pas possible en général d'obtenir un invariant fini. Un système

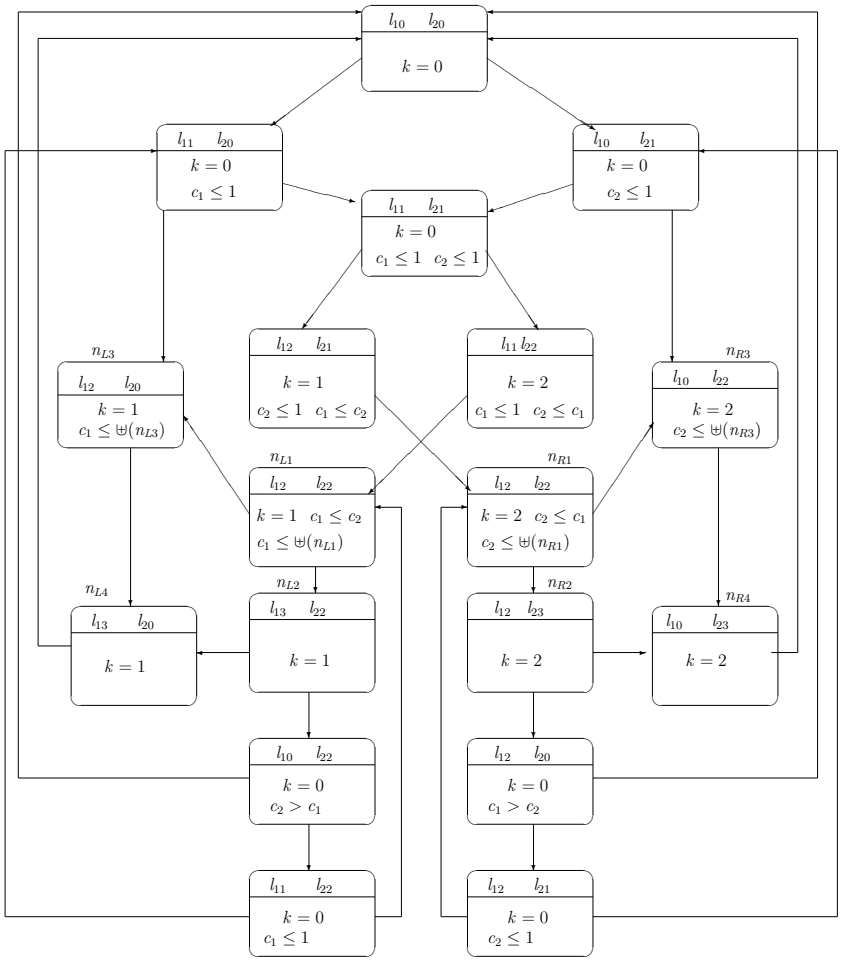


Fig. 1.7: *Diagramme Δ_2 pour le protocole de Fischer (voir fig. 1.4).*

paramétré typique consiste en un nombre *quelconque* de processus identiques qui interagissent au moyen d'une communication synchrone ou asynchrone. Des exemples typiques sont des protocoles d'exclusion mutuelle pour un nombre quelconque de processus en lice pour une ressource commune.

Des techniques conventionnelles de *model checking* peuvent être utilisées pour vérifier des instances de systèmes paramétrés pour des valeurs fixes du paramètre, par exemple pour un nombre donné de processus participant. Afin de démontrer la correction pour une valeur quelconque du paramètre, il faut construire un objet syntaxique unique qui représente (est une abstraction pour) la famille entière de systèmes. Nous allons discuter dans ce chapitre l'utilisation de PDT pour la vérification de systèmes paramétrés, prenant une version à n processus du protocole d'exclusion mutuelle de Fischer.

Nous considérons deux classes de propriétés : celles du système entier, c'est à dire concernant tous les processus, et des propriétés «par processus» qui doivent être vérifiées par tout processus du système. Ces dernières propriétés sont souvent appelées des propriétés *universelles*. Étant donné un système paramétré qui consiste en N processus, les propriétés universelles sont représentées par des formules de la forme $\forall k \in 1...N : P(k)$. Baukus et al. [13] ont étudié des techniques pour la vérification de propriétés universelles de systèmes paramétrés fondées sur la transformation d'une famille infinie de systèmes en un système de transition exprimé comme une formule de la logique WS1S, appliquant des techniques d'abstraction sur ce système.

XTG paramétrés

Nous commençons par définir une version paramétrée d'XTG. La structure de communications devient importante dans le cas où le système d'XTG contient des fonctions pour l'échange de valeurs et la synchronisation (voir les définitions 1.8 et 1.7).

Définition 1.16 (XTG paramétré)**.** *Un XTG paramétré consiste en un nombre quelconque mais fini de processus identiques avec m états qui interagissent par communication synchrone. Formellement, $\mathcal{X} = \langle Init, \langle P_1, \ldots, P_n \rangle, ch \rangle$ où chaque $P_i = \langle Init_i, L_i, l_{0i}, I_i, E_i, U_i \rangle$ est aussi un XTG. La sémantique de $\mathcal{X}$ est donnée par la définition 1.9.*

PDT paramétrés

Dans la définition 1.10 nous avons introduit les PDTs par rapport aux ensembles L d'états de contrôle et $\mathcal{P}$ de prédicats d'abstraction. Dans le contexte de systèmes paramétrés nous allons étendre $\mathcal{P}$ afin qu'il représente des prédicats paramétrés. Étant donné un XTG paramétré nous allons permettre dans ce qui suit que l'ensemble $\mathcal{P}$ contienne non seulement des prédicats habituels mais aussi des *prédicats paramétrés* par l'identifiant des processus.

Définition 1.17 (prédicats quantifiés)**.** *Un prédicat quantifié est d'une des deux formes*

- $\forall i \in 1, \ldots, N : f(i)$ *ou*
- $\exists i \in 1, \ldots, N : f(i)$

où $f(i)$ est une formule (non-temporelle) sans quantificateur avec un paramètre i qui représente l'identifiant d'un processus. L'ensemble des prédicats quantifiés sera dénoté par $\mathcal{P}_Q$.

Les prédicats quantifiés remplacent l'ensemble $\mathcal{P}$ des prédicats de base dans la définition 1.10 des PDT qui sont maintenant définis par rapport à un ensemble L d'états de contrôle et l'ensemble $\mathcal{P}_Q$. Comme auparavant nous dénotons par $\overline{\mathcal{P}_Q}$ l'ensemble des prédicats quantifiés et leurs négations.

Outre cette extension motivée par la vérification de systèmes paramétrés, la définition suivante de PDT paramétrés introduit également des annotations liées aux hypothèses d'équité et à des relations d'ordre. Pour cette dernière condition, nous supposons donné un ensemble $\mathcal{O}$ de relations d'ordre bien-fondées $\prec$. Par $\preceq$ nous dénotons la clôture réflexive de $\prec$, et par $\mathcal{O}^=$ l'ensemble des relations dans $\mathcal{O}$ et leurs clôtures réflexives. Ces extensions sont transposées littéralement des travaux originaux sur les diagrammes de prédicats pour des systèmes non-temporisés [17, 18]. Nous avons préféré ne pas les introduire dans la définition 1.10 originale des PDT pour ne pas l'encombrer et pour nous focaliser sur les aspects liés au temps réel.

Définition 1.18 (PDT paramétrés et leurs exécutions)**.** *Étant donnés des ensembles finis L et $\mathcal{P}_Q$, un* PDT paramétré *sur L et $\mathcal{P}_Q$ est un sextuple $\Delta_p = \langle N, N_0, R_\mu, R_\tau, o, \Theta \rangle$ comme suit :*

- *$N \subseteq L \times 2^{\overline{\mathcal{P}_Q}}$ est un ensemble fini de nœuds.*
- *$N_0 \subseteq N$ est l'ensemble des nœuds initiaux.*
- *$R_\mu, R_\tau \subseteq N \times N$ sont deux relations représentant les transitions discrètes et correspondant au passage de temps. La relation R_τ est supposée réflexive.*
- *o est un étiquetage associant un ensemble fini $\{(\eta_1, \prec_1), \ldots, (\eta_h, \prec_h)\}$ d'expressions η_i, couplées avec des relations $\prec_i \in \mathcal{O}^=$.*
- *$\Theta : R_\mu \to \{NF, WF, SF\}$ associe une hypothèse d'équité avec les transitions discrètes ; ces valeurs représentent respectivement pas d'équité, équité faible et équité forte.*

Une exécution *de Δ_p est une suite infinie*

$$\sigma \quad = \quad n_0 \xrightarrow{lab_0} n_1 \xrightarrow{lab_1} n_2 \ldots$$

où $n_0 \in N_0$, $lab_i \in \{\mu, \tau\}$ et $n_i \rightarrow_{lab_i} n_{i+1}$ pour tous $i \in \mathbb{N}$ vérifiant les conditions suivantes :

- *pour toute transition $(n, n') \in R_\mu$ telle que $\Theta(n, n') = WF$, soit la transition $n \xrightarrow{\mu} n'$ apparaît infiniment souvent dans σ, soit $n_i \neq n$ pour un ensemble infini de i,*

- *pour toute transition* $(n, n') \in R_\mu$ *telle que* $\Theta(n, n') = SF$*, soit la transition* $n \xrightarrow{\mu} n'$ *apparaît infiniment souvent dans* σ*, soit* $n_i = n$ *pour un ensemble fini seulement de* i*,*
- *pour tout couple* $(\eta, \prec)$ *d'une expression* η *et une relation irréflexive* $\prec$*, s'il existe un nombre infini de transitions* $n_i \xrightarrow{\mu} n_{i+1}$ *dans* σ *avec* $(\eta, \prec) \in o(n_i, n_{i+1})$ *alors il existe un nombre infini de transitions* $n_j \xrightarrow{\mu} n_{j+1}$ *dans* σ *avec* $(\eta, \prec) \notin o(n_j, n_{j+1})$ *et* $(\eta, \preceq) \notin o(n_j, n_{j+1})$*.*

Un PDT paramétré Δ_p est conforme à un XTG paramétré $\mathcal{X}_p$ si chaque exécution de (toutes les instances de) $\mathcal{X}_p$ a une trace par Δ_P. Le théorème 1.12 de conformité est étendu aux PDT paramétrés avec conditions d'équité et d'ordre. Outre le passage implicite de temps (formalisé par les nœuds bornant le temps), les hypothèses d'équité évitent qu'une exécution bégaie indéfiniment. Nous exigeons (voir [18]) que les transitions de l'XTG paramétré correspondant à du bégaiement dans Δ_p ou au passage de temps n'augmentent pas (par rapport à $\prec$) la valeur des expressions η telles que $(\eta, \prec)$ apparaît dans l'étiquette d'une transition sortant du nœud.

Théorème 1.19. *Soit* $\mathcal{X}_p = \langle Init, L, \vec{l_0}, I, E, U \rangle$ *un XTG paramétré. Le PDT paramétré* $\Delta_p = \langle N, N_0, R_\mu, R_\tau, o, \Theta \rangle$ *sur* L *et* $\mathcal{P}_Q$ *est conforme à* $\mathcal{X}_p$ *si toutes les conditions suivantes sont vraies :*

1. $Init \wedge I(\vec{l_0}) \Rightarrow \bigvee_{\langle \vec{l_0}, P \rangle \in N_0} P$

   La condition globale initiale de l'XTG et l'invariant associé à l'état initial impliquent le prédicat d'un nœud initial de Δ_p *associé à l'état initial de* $\mathcal{X}_p$*.*

2. *Pour tout nœud* $n = \langle \vec{l}, P \rangle$ *de* Δ_p *:*

   (a) *Pour toute arête* $e = \langle l, g, u, l' \rangle$ *d'un processus de* $\mathcal{X}_p$ *telle que* $l \in \vec{l}$ *et* $sync(e, e') = \bot$ *pour toute arête* $e' \in E$*, soit* V_u *l'ensemble des variables* v *mises à jour par* u *(c'est à dire tel que* $\langle v, e \rangle \in u$ *pour un certain* e*), et soit* N' *l'ensemble des nœuds* $n' = \langle \vec{l'}, Q \rangle$ *de* Δ_p *tel que* $\vec{l'}$ *résulte de* $\vec{l}$ *par la transition du processus en question de* l *à* l'*.*

   $$P \wedge g \wedge I(\vec{l}) \wedge I'(\vec{l'}) \wedge \Big(\bigwedge_{\langle v, e \rangle \in u} v' = e \Big) \wedge \Big(\bigwedge_{v \in V \setminus V_u} v' = v \Big) \Rightarrow \bigvee_{\langle \vec{l'}, Q \rangle \in N'} Q'$$

   Pour les transitions locales des processus qui ne se synchronisent avec aucune autre transition, l'étiquette du nœud n*, les invariants associés aux états actifs avant et après la transition de* $\mathcal{X}_p$*, ainsi que la mise à jour engendrée par celle-ci impliquent les prédicats d'un certain nœud de* N'*.*

(b) *Soient* $e_1 = \langle l_1, g_1, u_1, l_1' \rangle$ *et* $e_2 = \langle l_2, g_2, u_2, l_2' \rangle$ *deux arêtes de deux processus différents de* $\mathcal{X}_p$ *telles que* $l_1, l_2 \in vecl$ *et* $sync(e_1, e_2) \neq \perp$, *soit* V_u *l'ensemble des variables* v *mises à jour par* u_1 *ou* u_2 *ou telles que* $(v, e) \in sync(e_1, e_2)$, *et soit* N' *l'ensemble des nœuds* $n' = \langle \vec{l}', Q \rangle$ *de* Δ_p *tel que* $\vec{l}'$ *résulte de* $\vec{l}$ *par la transition conjointe des deux processus.*

$$\begin{aligned} & P \wedge g_1 \wedge g_2 \wedge I(\vec{l}) \wedge I'(\vec{l}') \\ & \wedge \left(\bigwedge_{\langle v, e \rangle \in u_1 \cup u_2} v' = e \right) \wedge \left(\bigwedge_{(v,e) \in sync(e_1, e_2)} v' = e \right) \wedge \left(\bigwedge_{v \in V \setminus V_u} v' = v \right) \\ \Rightarrow \quad & \bigvee_{\langle \vec{l}', Q \rangle \in N'} Q' \end{aligned}$$

Pour tout couple de transitions pouvant se synchroniser, l'étiquette du nœud n, *les invariants associés aux états actifs avant et après la transition conjointe, ainsi que les mises à jour engendrées localement par chacune des transitions et par l'échange de valeurs entre les processus impliquent les prédicats d'un certain nœud de* N'.

3. *Pour tout nœud* $n = \langle \vec{l}, P \rangle$ *de* Δ_p, *soit* N'' *l'ensemble des nœuds* $n'' = \langle \vec{l}, Q \rangle$ *associés au même vecteur d'états de contrôle que* n *et tels que* $n \rightarrow_\tau n''$.

$$\begin{aligned} & P \wedge \delta \in \mathbb{R}^{\geq 0} \wedge {}_{c \in V_c} c' = c + \delta \wedge \bigwedge_{v \in V \setminus V_c} v' = v \wedge I(\vec{l}) \wedge I'(\vec{l}) \\ & \wedge \forall \varepsilon \leq \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow\; I''(\vec{l}) \\ & \wedge \forall \varepsilon < \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow\; \bigwedge_{\langle \vec{l}, g, u, \vec{l}' \rangle \in U} \neg g'' \\ \Rightarrow \quad & \bigvee_{\langle \vec{l}, Q \rangle \in N''} Q' \end{aligned}$$

Pour toute transition correspondant à un passage de temps par δ *unités et telle qu'aucune arête urgente de* $\mathcal{X}_p$ *partant de l'état conjoint de* n *n'est activée avant, l'étiquette de* n *et les invariants des états actifs dans* n *(pour toutes les valeurs de temps intermédiaires) impliquent que l'étiquette d'un certain nœud de* N'' *soit vérifiée.*

4. *Pour tout nœud* $n = \langle \vec{l}, P \rangle$ *de* Δ_p :

   (a) *pour tout nœud* $n' = \langle \vec{l}', Q \rangle$ *de* Δ_p *avec* $n \rightarrow_\mu n'$

      i. *pour toute arête* $e = \langle l, g, u, l' \rangle$ *dont la transition correspond aux états de contrôle* $\vec{l}$ *et* $\vec{l}'$ *et telle que* $sync(e, e') = \perp$ *pour toute arête*

$e' \in E$ et avec les mêmes notations que dans la condition (2a) :

$$\begin{array}{rl} & P \wedge g \wedge I(\vec{l}) \wedge I'(\vec{l}') \wedge Q' \\ & \wedge \left(\bigwedge_{\langle v,e\rangle \in u} v' = e \right) \wedge \left(\bigwedge_{v \in V \setminus V_u} v' = v \right) \\ \Rightarrow & \bigwedge_{(\eta,\prec) \in o(n,n')} \eta' \prec \eta \end{array}$$

ii. pour tout couple d'arêtes $e_1 = \langle l_1, g_1, u_1, l_1' \rangle$ et $e_2 = \langle l_2, g_2, u_2, l_2' \rangle$ comme dans la condition (2b) et avec les mêmes notations :

$$\begin{array}{rl} & P \wedge g_1 \wedge g_2 \wedge I(\vec{l}) \wedge I'(\vec{l}') \wedge Q' \\ & \wedge \left(\bigwedge_{\langle v,e\rangle \in u_1 \cup u_2} v' = e \right) \wedge \left(\bigwedge_{(v,e) \in sync(e_1,e_2)} v' = e \right) \wedge \left(\bigwedge_{v \in V \setminus V_u} v' = v \right) \\ \Rightarrow & \bigwedge_{(\eta,\prec) \in o(n,n')} \eta' \prec \eta \end{array}$$

(b) pour tout nœud $n'' = \langle \vec{l}'', Q \rangle$ de Δ_p avec $n \rightarrow_\tau n''$ et toute transition $n \rightarrow_\mu n'$ dans Δ_p :

$$\begin{array}{rl} & P \wedge \delta \in \mathbb{R}^{\geq 0} \wedge {}_{c \in V_c} c' = c + \delta \wedge \bigwedge_{v \in V \setminus V_c} v' = v \wedge I(\vec{l}) \wedge I'(\vec{l}'') \wedge Q' \\ & \wedge \forall \varepsilon \leq \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow\; I''(\vec{l}) \\ & \wedge \forall \varepsilon < \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow\; \bigwedge_{\langle \vec{l},g,u,\vec{l}' \rangle \in U} \neg g'' \\ \Rightarrow & \bigwedge_{(\eta,\prec) \in o(n,n')} \eta' \preceq \eta \end{array}$$

Vérification de propriétés concernant le système entier

Nous démontrons l'utilisation de PDT paramétrés pour la vérification de propriétés «globales» concernant le système entier, toujours pour l'exemple du protocole de Fischer.

Nous écrivons $p[i].c$ et $p[i].loc$ pour l'horloge locale et l'état de contrôle local au processus i. Par cs nous dénotons l'ensemble des processus se trouvant dans leur section critique, c'est à dire

$$cs \;=\; \{i \in 1..N : p[i].loc = 3\}$$

Comme pour la version à deux processus, le système contient une variable partagée k dont la valeur indique le processus pouvant entrer en section critique. Comme auparavant, la démarche globale de vérification consiste en deux étapes en démontrant

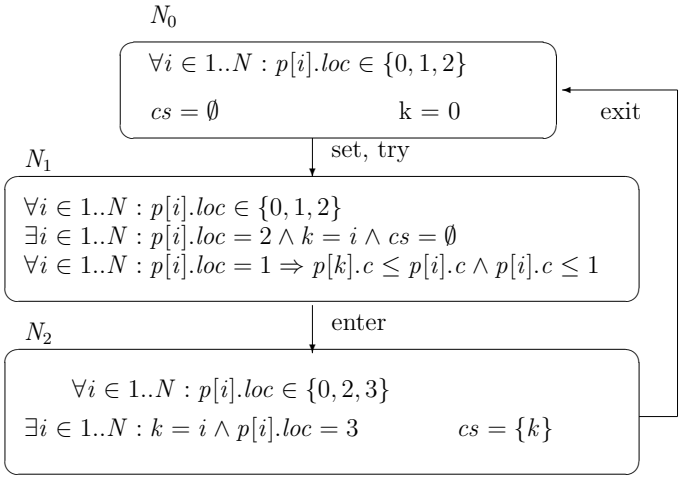


Fig. 1.8: *Le diagramme $\Delta 1_p$ pour le protocole de Fischer à N processus*

d'abord la conformité de Δ_p par rapport à $\mathcal{X}_p$ et puis en vérifiant des propriétés du diagramme fini Δ_p en utilisant le *model checking*.

La figure 1.8 montre le diagramme $\Delta 1_p$ pour le protocole de Fischer pour n processus que l'on démontre être conforme à $\mathcal{X}_p$ en vérifiant les conditions du théorème 1.19. Les arêtes portent des noms qui correspondent aux transitions de $\mathcal{X}_p$.

Notre intérêt se focalise sur les transitions du processus k (c'est à dire dont l'identifiant est indiqué par la variable partagée k) : après qu'un certain processus i prenne la transition de l'état 1 à l'état 2 il devient le processus k car son identifiant est enregistré dans la variable k tout en remettant l'horloge locale à 0. Les autres processus se trouvant à l'état 1 vérifient la contrainte $c \leq 1$ et attendent la transition à l'état 2 ; les valeurs de leurs horloges deviennent alors inférieures à l'horloge du processus k qui est remise à 0. Le prédicat correspondant

$$\forall i \in 1..N : p[i].loc = 1 \Rightarrow (p[k].c \leq p[i].c) \wedge (p[i].c \leq 1)$$

est indiqué dans le nœud N_1.

Les autres obligations de preuve sont démontrées de manière similaire et CVC-Lite les démontre avec succès. Ces obligations de preuve et le code pour CVC-Lite apparaissent dans l'annexe A. (Encore une fois, les seules τ-transitions du diagramme $\Delta 1_p$ de la figure 1.8 sont les boucles implicites sur les nœuds.)

Considérant $\Delta 1_p$ comme un système fini de transitions étiquetées, ses exécutions peuvent être codées dans les langages de modélisation d'outils standard de

model checking. Nous ajoutons des variables spéciales à $\Delta 1_p$ afin de vérifier la propriété d'exclusion mutuelle exprimant que deux processus ne peuvent jamais se trouver en même temps dans leurs sections critiques :

$$\Box(\forall i, j \in 1..N : i \in cs \wedge j \in cs \Rightarrow i = j).$$

Cette fois il ne suffit pas tout à fait de considérer les prédicats comme des variables booléennes et d'utiliser des *model checker* ordinaires car l'inférence de l'exclusion mutuelle à partir des étiquettes des nœuds requiert l'utilisation de la théorie élémentaire des ensembles. Le *model checking* suffirait à condition d'ajouter des prédicats supplémentaires aux nœuds de $\Delta 1_p$.

De manière alternative nous pouvons nous servir de CVC-Lite afin de déduire la propriété de l'exclusion mutuelle, par exemple en vérifiant que chaque nœud de $\Delta 1_p$ vérifie la formule

$$cs = \emptyset \vee cs = \{k\}$$

ce qui est aisé à prouver pour CVC-Lite.

Vérification de propriétés universelles

Le diagramme $\Delta 1_p$ nous a été utile pour vérifier la propriété d'exclusion mutuelle. Cependant la démarche que nous avons utilisée pour la vérification de propriétés concernant le système entier ne peut pas être utilisée afin de démontrer l'accessibilité individuelle pour ce protocole car elle ne nous permet pas de tracer le comportement d'un processus spécifique. En général, cette démarche présente la limitation qu'elle ne peut être utilisée pour la vérification de propriétés universelles.

Les propriétés de systèmes paramétrés s'expriment souvent à travers des formules de la forme $\forall i \in 1..N : P(i)$ où la formule LTL $P(i)$ décrit une propriété d'un processus seul. De telles propriétés peuvent être vérifiées en distinguant un processus spécifique $i \in 1..N$ des autres processus. De manière formelle, cette approche correspond à l'introduction d'une constante de Skolem i et la preuve de la propriété $i \in 1..N \Rightarrow P(i)$.

L'idée derrière la construction d'un PDT paramétré pour une propriété *universelle* est de suivre l'évolution du processus i et de représenter les transitions des autres processus dans une partie à part du diagramme. Le diagramme $\Delta 2_p$ de la figure 1.9 illustre cette idée pour la version à N processus du protocole de Fischer. Afin de simplifier les étiquettes des nœuds nous écrivons $\forall Q(j)$ comme une abbréviation de la formule

$$\forall j \in 1..N \setminus \{i\} : Q(j)$$

et de manière similaire pour $\exists Q(j)$.

$\Delta 2_p$ est une abstraction correcte de la version à N processus du protocole de Fischer. Nous faisons encore une fois appel au théorème 1.19 afin de démontrer cette relation de conformité, et la preuve apparaît dans l'annexe A.

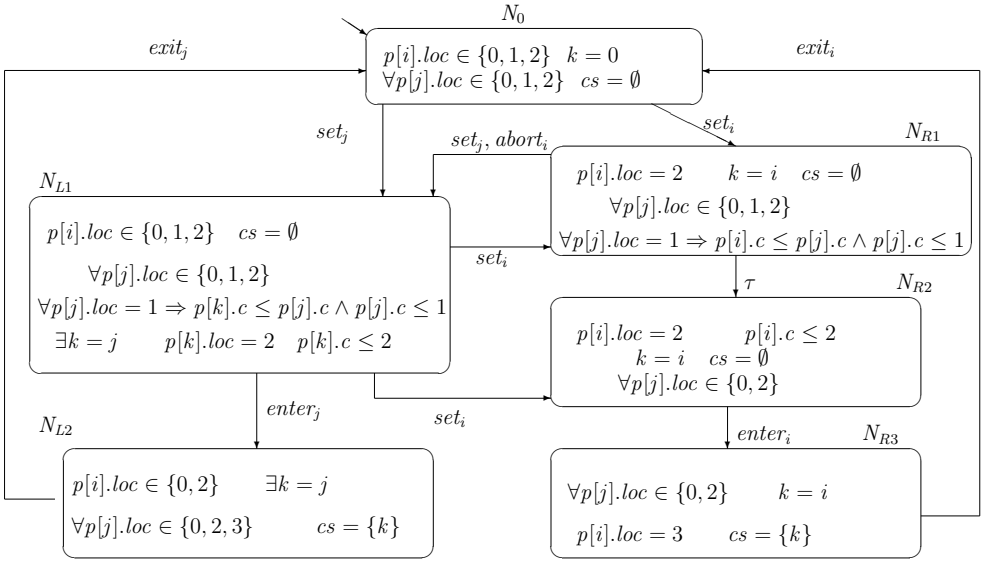


Fig. 1.9: $\Delta 2_p$ *pour le protocole de Fischer pour N processus.*

Les nœuds N_{R1}, N_{R2} et N_{L1} du diagramme $\Delta 2_p$ correspondent aux situations où soit le processus i, soit un autre processus j se trouve à l'état l_2. Considérons d'abord N_{R1} et regardons la transition vers N_{R2} : après qu'un certain processus i prenne la transition set_i à partir de l'état l_1 alors ce processus reste à l'état l_2 jusqu'à ce que tous les autres processus j soient arrivés à l'état l_2.

Considérons maintenant la transition de N_{R1} vers N_{L1} étiquetée set_i, $abort_j$. Après qu'un certain processus ait pris la transition set_j (menant de N_{L0} à N_{L1}) il devient le processus k et remet son horloge locale à 0. Les horloges de tous les autres processus se trouvant à l'état l_1 continuent à évoluer normalement ce qui justifie l'ajout du prédicat

$$\forall j \in 1..N : p[j].loc = 1 \Rightarrow (p[k].c \leq p[j].c) \wedge (p[j].c \leq 1)$$

à l'étiquette du nœud N_{L1}.

Pendant que le processus k attend de passer à l'état l_3, le processus i peut arriver à l'état l_2 et devenir le processus gagnant (c'est à dire, enregistrer i dans la variable k). Cette transition correspond aux arêtes de N_{L1} à N_{R1} ou à N_{R2}, selon le fait qu'il y ait d'autres processus à l'état l_1 ou non. En particulier, le nœud N_{R2} représente le «changement du processus gagnant».

Le diagramme $\Delta 2_p$ nous permet de démontrer la propriété de vivacité du protocole qui énonce qu'un des processus finira par accéder à la section critique,

c'est à dire on vérifiera la formule LTL

$$\forall i \in 1...N : \Box\big((k = i \wedge p[k].loc = 2) \Rightarrow \Diamond(p[k].loc = 3)\big)$$

Aussi, la propriété d'exclusion mutuelle du protocole de Fischer pour N processus peut être vérifiée à partir du diagramme $\Delta 2_p$ de la figure 1.9. A l'aide de CVC-Lite nous pouvons déduire la formule alternative

$$\forall i \in 1, \ldots, N : cs = \emptyset \vee cs = \{k\}$$

à partir de l'étiquette de chaque nœud de $\Delta 2_p$. Nous pouvons ainsi dire de manière générale que les PDT paramétrés sont aussi utiles pour la vérification de propriétés universelles de $\mathcal{X}_p$ que pour la vérification de propriétés concernant le système entier.

1.5 Conclusions

Cette thèse a présenté une technique pour la vérification formelle de systèmes temps-réel qui est fondée sur la combinaison de techniques différentes telles que l'abstraction booléenne, la démonstration de théorèmes et le *model checking*. Cette technique, que nous avons appelé IAR, permet d'intégrer dans HOL une technique de représentation abstraite booléenne qui tire des éléments de technologies SAT et de *model checking*.

Nous l'avons développée en considérant successivement trois aspects clefs que nous avons identifiés dans l'introduction : la représentation abstraite, le raffinement des abstractions et l'analyse de l'abstraction finie. Pour chacun des aspects nous avons proposé une solution, fondée sur l'hypothèse de base que l'aspect de représentation abstraite est fondée sur une manière d'abstraction qui réduit le nombre d'états du modèle concret donné en faisant abstraction de comportements qui ne sont pas essentiels à la vérification par IAR.

Nous faisons appel à divers outils pour supporter notre méthodologie IAR : les domaines concernés sont mentionnés en correspondance avec chacun des techniques et outils dans l'énumération suivante :

- modélisation de systèmes (XTG : graphe temporisé étendu, *extended time graph*, PDT : diagrammes de prédicats pour systèmes temporisés, *predicate diagrams for timed systems*),
- abstraction et raffinement (DIXIT : boîte à outils visuelle pour l'abstraction booléenne, CVC-Lite : démonstrateur de théorèmes),
- vérification (LPMC et Spin : *model checker*, CVC-Lite : vérificateur de validité).

La démarche IAR décrite dans une perspective d'évaluation vise la vérification de propriétés en logique temporelle LTL, sur une abstraction correcte sous la forme d'une représentation abstraite conforme à un automate temporisé étendu.

Le problème de vérification peut alors être réduit à deux sous-problèmes qui sont (a) d'établir la correction de la représentation abstraite et (b) d'évaluer

la propriété d'intérêt sur la représentation abstraite. Afin d'évaluer l'utilité du concept d'IAR et la mise en œuvre des deux sous-problèmes nous l'avons appliqué à quelques exemples.

Nous avons étudié le sous-problème (a) sous la forme de la vérification de la conformité. Une présentation générale de la démarche apparaît dans la figure 1.3, et les solutions aux deux sous-problèmes sont expliquées dans les sections 1.2 et 1.3. Étant donnés un ensemble de prédicats pour l'abstraction, des propriétés d'intérêt et la description du système concret nous construisons d'abord à la main une représentation abstraite approximative (représentation abstraite réduite). Ensuite, la conformité de cette représentation réduite est évaluée afin de l'affiner de manière adéquate : si l'analyse de conformité renvoie le résultat «incorrect» alors nous affinons successivement l'abstraction jusqu'à ce qu'elle devienne «correcte».

En même temps IAR considère aussi si les propriétés d'intérêt sont vérifiables sur la représentation abstraite. Si les deux conditions (conformité et vérification) sont satisfaites pour la représentation abstraite nous l'appelons une représentation abstraite complète, impliquant que les propriétés LTL qui sont vérifiées par le modèle abstrait le sont aussi par le modèle concret. Sinon, le modèle abstrait n'est pas complet et nous devons le concrétiser (affiner) jusqu'à ce qu'il devienne complet.

La méthode IAR termine lorsque les conditions de vérification pour la conformité sont vérifiées et que la procédure ne trouve pas d'autres nœuds et arêtes à ajouter à la représentation abstraite actuelle.

Hormis une meilleure illustration et implémentation de la procédure IAR nous avons proposé le format des diagrammes de prédicats pour les systèmes temporisés (PDT) qui est une notation pour la représentation d'abstractions booléennes de systèmes temporisés. Ce format est une variante des diagrammes de prédicats pour les systèmes discrets, en distinguant les transitions de passage de temps des transitions discrètes. Nous avons également établi un ensemble d'obligations de preuve pour démontrer la conformité entre un modèle XTG d'un système temporisé ; ces obligations sont induites par les conditions des théorèmes que nous avons proposés. Ces conditions, sous forme de formules de la logique de premier ordre, peuvent aussi être utiles afin de trouver la bonne représentation abstraite et d'identifier les prédicats pour l'abstraction en analysant les obligations de preuve engendrées par la procédure IAR. Les obligations de vérification peuvent être démontrées en utilisant des outils automatiques de preuve pour certaines théories de premier ordre, tels que les solveurs SAT ou CVC-Lite.

En ce qui concerne l'évaluation, le problème de décision pour LTL a été réécrit en un problème de satisfiabilité booléenne (SAT) de manière à ce que la propriété LTL pouvait être traduite en une autre formule dont la validité impliquait que la formule originale était vraie. Il est d'une importance égale de déterminer que de telles affectations n'existent pas, impliquant que la propriété exprimée par la formule alternative égale faux pour toutes les évaluations possibles des variables. Dans ce dernier cas nous disons que la propriété est insatisfaisable sur les PDT ; sinon elle est satisfaisable. CVC-Lite a été utilisé afin de déterminer la satisfia-

bilité de cette formule alternative.

Dans ce sens, les PDT constituent une interface entre les techniques de vérification fondées sur la déduction et sur le *model checking*. Les nœuds des PDT sont étiquetés par des prédicats qui sont interprétés lors de la vérification de la conformité mais sont (essentiellement) considérés comme des variables booléennes lors du *model checking*. Le format des diagrammes de prédicat est supporté par la boîte à outils DIXIT, et nous avons démontré son utilisation dans le cadre d'une démarche IAR à travers le protocole d'exclusion mutuelle de Fischer et d'un système de passage à niveau. Nous avons aussi montré que la démarche fondée sur les PDT est utile non seulement pour la vérification de propriétés de sûreté mais aussi de vivacité.

Les travaux décrits dans le chapitre 1.3 ont été étendus à des modèles plus riches, comme les systèmes paramétrés. Plus spécifiquement, notre intérêt primitif en des termes de *model checking* est de pouvoir contourner les limitations intrinsèques de ces techniques concernant la modélisation de données et qui apparaissent dans les outils de vérification tels que PMC et Uppaal qui prennent des XTG ou autres formes d'automates temporisés comme langage d'entrée.

Nous avons démontré dans la section 1.4.2 que le format des PDT se prête également à des systèmes paramétrés comme la version du protocole de Fischer pour N processus. Pour ces applications, toute une famille de processus est représentée dans un seul diagramme. En effet, nous sommes souvent intéressés par des propriétés *universelles* de systèmes paramétrés, et elles peuvent être établies en identifiant un processus spécifique et en suivant son comportement séparément de celui du reste du système.

Dans le chapitre 6 (uniquement présent dans la partie en anglais) nous avons utilisé une étude de cas d'un passage à niveau afin de démontrer pleinement que la démarche IAR nous permet de générer des PDT complets. En particulier nous avons utilisé des itérations de techniques de preuve pendant la construction des PDT. Cette étude de cas a également été étendue afin de traiter un système de passage à niveau paramétré.

1.6 Travaux futurs

Nous voyons ce travail comme un premier pas vers l'application d'abstractions booléennes pour la vérification de systèmes temporisés. Une des limitations actuelles réside dans le fait que nous faisons abstraction du temps précis qui passe dans une transition de ce type. Ainsi, nous ne pouvons pas vérifier facilement des propriétés quantitatives, comme des bornes supérieures sur les temps de réponse, bien que des propriétés mentionnant des horloges individuelles puissent être vérifiées. Nous voudrions étudier deux solutions possibles à ce problème, soit en utilisant une logique temporelle temps-réel (TLTL), soit en introduisant des horloges auxiliaires pendant la vérification, comme l'ont proposé Henzinger et al. [34] et Tripakis [57]. Ceci nous permettrait en particulier de bénéficier d'outils

de *model checking* pour les systèmes temporisés comme Uppaal [6] ou PMC.

Aussi visons-nous à réduire le nombre de conditions de vérification que les utilisateurs ont à établir à l'aide d'un outil de preuve afin d'établir la conformité. En effet, nous considérons les conditions des théorèmes 1.12 et 1.19 comme établissant les conditions extrêmes qu'un PDT doit satisfaire, et nous observons que la plupart de celles-ci sont assez triviales pour des exemples typiques. Il sera intéressant de nous limiter à des classes spécifiques de systèmes qui engendrent des obligations de preuve décidables, ainsi permettant la construction automatique des PDT.

Nous visons également une étude plus poussée de techniques de raffinement pour la construction des PDT, étant donné un XTG et un ensemble de prédicats d'abstraction. Des travaux préliminaires concernant la combinaison d'outils pour l'interprétation abstraite et l'exploration d'états ont été décrits dans [38, 39], mais il nous faudra plus d'expérience afin d'identifier des abstractions complètes pour des systèmes temporisés.

Bien que l'abstraction booléenne de systèmes temporisés ait été étendue à des modèles plus riches tels que les automates temporisés paramétrés et même à des automates temporisés comportant d'autres types infinis comme les compteurs ou les piles, le prix à payer est évidemment que de telles extensions sont indécidables par nécessité. Dans des travaux futurs nous aimerions étudier des formules faisant abstraction du temps réel impliquant de l'arithmétique et d'autres contraintes au-delà des seules variables booléennes. De cette manière nous pourrions exprimer et vérifier d'autres propriétés intéressantes, comme par exemple des bornes sur les temps de réponse.

Nous avons rencontré quelques problèmes dûs à l'incomplétude des solveurs SMT pour certaines théories. Par exemple, CVC-Lite n'est pas capable de traiter des formules complexes de combinaisons riches de théories, en particulier concernant de nombreuses instanciations de quantificateurs, et nous pourrions essayer d'autres outils de preuve pour contourner ces limitations de CVC-Lite. Néanmoins, l'application de types différents de solveurs SMT pose des problèmes supplémentaires aux utilisateurs qui sont contraints à apprendre les commandes et tactiques des différents outils ; aussi, l'utilisation d'outils automatiques est limitée par les domaines que ces derniers supportent et qui peuvent ne pas être assez riches pour les théories mathématiques dans lesquelles travaillent les utilisateurs.

De manière idéale on aimerait disposer d'outils dédiés qui supportent des formules de grande complexité propositionnelle en des théories expressives et avec une interaction limitée avec les utilisateurs, ainsi qu'améliorer le degré d'automatisation d'assistants interactifs de preuve en les intégrant avec des outils automatiques.

Chapter 2

Introduction

Computer systems are more and more becoming an indispensable part of our daily life. As a consequence, our society has become highly dependent on computer hardware and software. This includes computer systems of which incorrect behaviour can have disaster consequences. Often, this concerns called safety-critical systems, for example nuclear controllers and certain type of medical systems, but also systems in which errors can bring enormous costs, like mass-produced microprocessors and certain types of financial systems.

Also, computer systems are becoming increasingly complex. Advanced technology in hardware have made it possible to build amazingly large and complex systems. This is particulary true for software systems, for which – due to the rapid increase in hardware performance – limitations on size are quickly becoming less relevant. This has resulted in a situation in which building correct complex software systems is almost impossible. The best that can be done is ensuring that for certain critical aspects, the correctness can shown to be likely. For hardware systems, potential complexity is somewhat more limited, but still correctness cannot always be guaranteed. In fact, we have seen numerous examples of design errors with impact consequences. The Ariane 5 crash due to a software error and the Pentium processor hardware error are perfect examples.

In fact, it is not possible to guarantee a reasonable level of correctness for critical systems, gives rise to very low expectations concerning the level of correctness of less critical systems. We can say that building correct software and hardware systems is still a major challenge – even more if one also takes into account other important characteristic dimensions of the development process, like development costs and longevity. Therefore, it is not surprising that a lot of attention has been paid to develop technologies and tools that focus on improving the quality of software and hardware, particulary those aiming at avoiding critical and costly errors. This results in the wide-spread usage of verification approaches like testing and simulation. The effort dedicated to verification is higher than the

effort involved in actually building the system. Also, in computer science there has been quite an effort to improving the building process of software, through the introduction of a wide range of methods and techniques aimed at ensuring that the resulting computer systems have an acceptable degree of correctness.

Formal methods were introduced more than twenty years ago as a potential approach towards building rigorously correct systems. The field of formal methods approaches to designing software and hardware systems that are firmly found on mathematics. The basic idea is that if systems are designed in terms of mathematically-based formalisms then a basis is created for building systems that are "provable" correct. Despite of much research, formal methods have not yet been adopted by the software industry.

However, in recent years an approach in formal methods has emerged which seems to have the potential to play a significant role in industrial software and hardware development practice. This thesis is developed based on this particular filed in formal methods, a formal verification approach called model checking. Model checking is an automated technique that can be used to verify formal specifications of hardware or software systems with formal specifications of properties. It is based on exhaustive exploration of state spaces. Its strength comes from the fact it can be done automatically without any complex human interaction. However, the major limitation is in its scalability – state spaces tend to explode as systems become more complex. Therefore, the major challenge in model checking is the search for techniques that allow efficient exploration of large state spaces.

This thesis addresses a certain class of verification problems, namely those that involve current real time systems in which quantitative timing aspects play a relevant role such as a class of systems in which formal verification is particular relevant. First of all, real time concurrent systems (often called real-time systems in this thesis) have an inherent complexity – due to the non-determinism introduced but the specification of parallel execution – that makes it very hard to ensure correctness of even quite simple systems. Also, timing aspects play an essential role for many safety-critical systems. Additionally, such timing aspects introduce a high level of complexity due to the fact that the behavior of timed system results in an infinite number of states because of unbounded parameters (time values) as often called states space explosion problem. In order to cope with the complexity of real-time systems verification, this thesis proposes a methodology which is a tool supported verification technique based on combination of abstraction, deductive and model-based verifications.

2.1 Background

Before defining the scope of this thesis, some background is provided. More elaborate introductions to formal verification such as predicate abstraction, deductive verification and model-based verification for real-time systems are described.

2.1.1 Formal Verification

Formal verification usually means that one is interested in the validity of some correctness statement about one or more formal specifications. This could for example be a completeness criterion (are all situations covered?), or a safety criterion (can something bad happen?). Often, a correctness statement involves another formal specification, at a higher level of abstraction. In that case, a verification problem consists of two specifications, one specifying requirements that the the other must satisfy. The nature of the relation between the two specifications differs. It could be that one states a very high-level property that the other must satisfy, or that a refinement relation exists between the two specifications. The two specifications can use the same specification languages or could use two different specification languages having a common semantic foundation.

Two fundamental techniques to verification can be discerned, namely deductive verification and model-based verification. Deductive verification – often referred to as theorem proving – is the more traditional approach to verification. Deductive techniques can in principle be used to verify infinite-state systems, based on sets of axioms and inference rules. Although these can be supported by theorem provers and interactive proof assistants, their use requires considerable expertise and tedious user interaction [4].

Model-based verification (often called just model checking in this thesis) on the other hand, does typically not have this disadvantage of deductive verification. The advantage is that verification becomes largely automated. However, in particular case such as real-time model checking then this verification is applicable only under certain restrictions; most notably, it requires the system to be represented as a timed automaton whose discrete state space (disregarding the real-valued clocks) is finite. This restriction is in general not satisfied for software systems, and ad-hoc approximations are therefore used in model checking. In this thesis, we use a new concept, named *iterative abstract refinement* for such approximations.

As saw above, the two approaches to formal verification are often considered as complementary, each having its own balance between level of automation and complexity of verification problems. Also, combinations of the two approaches with appropriate approximations should give rise to powerful verification environments. For example, a theorem prover can be used to verify that a finite-state model is a correct abstraction of a given system, and properties of that finite-state abstraction can then be established using model checking. In order to make this idea more concrete, we need to identify a suitable format that serves as an interface between abstract, deductive and model-based verification techniques and that gives rise to feasible verification conditions.

Tooling is an indispensable aspect of any industrially useful formal methods. Formal verification without tool support implies that it relies on correctness of human reasoning and documentation, which would be much in conflict with the goals of formal methods. Therefore, sophisticated tool support can be seen as an

inevitable factors for formal verification.

2.1.2 Abstraction and Refinement Techniques

This thesis presents a methodology for the verification of real-time systems using abstraction and refinement techniques whose idea is to reduce the number of states of a model by abstracting away behaviors that are not essential to the verification. The generic techniques are known as incremental abstraction refinement [2].

The proposed abstract-refinement framework in the thesis is mainly based on the existing abstraction method, such as predicate abstraction. Predicate abstraction [30, 59] has emerged as a fruitful basis for software verification and underlied tools such as SLAM [10] and BLAST [36].

Predicate abstraction also has been found to be a powerful tool for software verification, and we transfer this idea to the domain of real-time systems. The basic assumption underlying predicate abstraction is that for the verification of a given property, the state space of a real system can be partitioned into finitely many equivalence classes. For example, the precise amount of time elapsed in a transition does not really matter as long as the clock values are within certain bounds and similarly, the precise values of the data can be abstracted with the help of predicates that indicate characteristic properties.

In general, the relationship between a real system (often called concrete model in this thesis) and an abstract model that underlies abstract interpretation is described by a Galois connection [22]. The abstract domain is a Boolean lattice whose atoms are the set of predicates true or false of a set of states in a concrete model. The model obtained by abstraction w.r.t. this lattice is an over-approximation that it includes all runs of the concrete system, but may also exhibit some behaviors that have no counterpart in the concrete system.

Those above approaches have inspired our own verification methodology, called IAR (Iterative-Abstract-Refinement) in [39], which constructs a correct abstract model of a given concrete system in order to handle the complexity of real-time systems verification.

The main goal of IAR is to show that for any specification of concrete system and any formula of the temporal propositional logic, if the specification of concrete system implies the formula, then the implication can be proven by a suitable abstract model; given abstraction mappings for each of the domains of the state space, a corresponding abstract model is computed in a way that each path (behaviour) in the concrete system has a "corresponding" path in the abstract model. Properties are expressed as formulas over a logic in which the existence of certain path cannot be expressed. For instance, the linear time logic LTL only allows for formulas which state that a property holds for all paths of the system. Soundness then is ensured by the fact that whenever a formula holds in the abstract model, it also holds in the concrete system.

2.2 Scope of the thesis

This thesis deals with three topics such as abstraction, deduction and model checking, applied to the verification problem of given systems. One assumption is that model checking is used to establish the validity of properties over the abstract models.

Before getting into deeply about the core topic of this thesis, we first give a general sense of model checking, then our goal is proposed as applying new concepts of verification in order to carry out a particular task for the verification: model checking a system (in the narrow sense) involves two disctinct phases, depicted in Figure 2.1.(a). First, a given notation S of the system has to be "unfolded" into a model C – this is called *(model) construction*. This is formalized by a mapping R called *representation*. Second, the property ϕ of interest has to be checked over this model: $C \models \phi$.

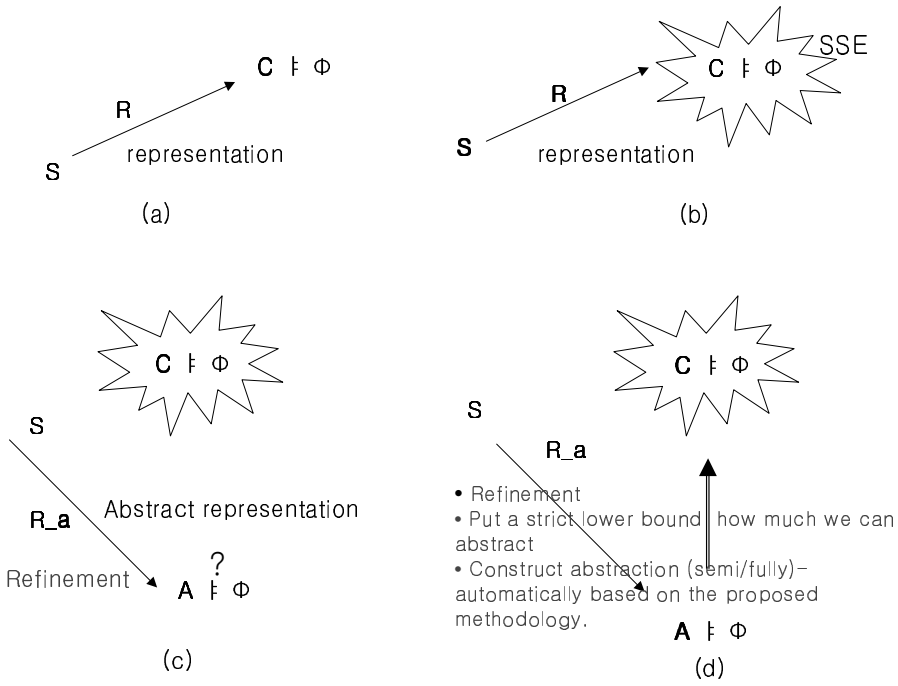


Figure 2.1: *Model checking and abstraction*

It is no surprise that in applying this method, early mentioned state-explosion problem, visualized in Figure 2.1.(b), forms the limiting factor.

Preferably, we would like to have a direct mapping from a concrete system notation S to the abstract model A and we call this mapping "*abstract repre-*

sentation" denoted R_a. As mentioned before, the number of states of model behaviours not essential to the verification may be abstracted in such a way that the size of the model is drastically reduced, then we consider if the abstract model is decidable for a property. This is illustrated in Figure 2.1.(c).

Ideally the model should be reduced to an abstraction A on which an efficient check can be performed, but under the condition that satisfaction of ϕ over A implies satisfaction over C. Note that we do not require that negative results carry over as well: if ϕ is not satisfied over A, then this does not imply that this is also the case for C. It may equally well be that too much information was abstracted away from the concrete model. See Figure 2.1.(d).

In a nutshell, our main goal is to represent a given complex system into a simple one, which can handle the limited scalability of model checking in practice as being impossible to explore entire state space with limited resources of time and memory. In fact model checkers are typically applied to models, which are abstracted away from details. In order too generat appropriate abstract models, some supports are required. Therefore, we present techniques for constructing such an appropriate model, which is simple and correct enough (conformed to the given system) to be used as input for verification with limited guidance from humans.

Precisely the construction of A directly from the S, is central in the theory of abstract representation. The first part of this thesis – Chapter 3 and Chapter 4 – deals with these aspects. Chapter 3 presents general theories of modelling systems. Chapter 4 offers systematic overview of the theory of abstract representation and presents evaluation of properties; a notion of abstract representation of systems (Predicate Diagrams for Timed systems: PDT) is introduced that aims to show that any property expressed in the specification logic LTL can be inferred from the PDT is guaranteed to hold in the PDT. In case the PDT conforms to the concrete system and preserves properties we desire to verify, this PDT considered as a *complete* abstract representation (*complete* model) of a concrete system.

Constructing the complete PDT is the core issue of this thesis: Chapter 5 describes how to construct such a complete PDT in terms of IAR metholdogy and how to verify safety and liveness property using the PDT as well.

Application of PDT is the main theme of the second part of this thesis: Chapter 6 shows the PDT can be extended to verify parameterized systems. In chapter 8 we add more explainations about how to generate a PDT, which can be viewed as a complete abstract representation of the systems being considered and show this PDT is relevant to the verification goal at the hand of once case study.

2.2.1 Key aspects of this thesis

The thesis focuses on the application (IAR) of abstract refinement techniques in verification of real-time systems. This IAR is developed by subsequently considering three key aspects: abstract representation, abstract refinement, and evaluation of given abstract representation. Although these aspects cannot always be viewed

in isolation, it is useful to consider them as different aspects of a verification problem:

1. Abstract Representation: Given a system specification – which is in fact an implicit description of a given system (concrete model) – IAR represents the state space of the concrete model in an abstraction manner, i.e. by reducing the number of states of the concrete model by abstracting away behaviors not essential to the verification.

2. Abstract Refinement: IAR guarantees that the representation is a correct abstraction of a concrete model. A theorem prover is employed to establish a correspondence between the concrete and abstract system. Although theorem provers typically require substantial user interaction and expertise, we expect the proof obligations for the theorem prover to be relatively easy, making our approach feasible in verification for infinite systems.

3. Evaluation (Verification): this mechanism should allow the model checker to decide applying refinement on an abstract representation by checking properties over the abstract representation is decidable or not.

The above distinction between three aspects will be used throughout this thesis, for each aspect, a solution is developed, with the basic assumption that the abstract representation and evaluation aspects are based on abstract-refinement. Thus, one could say that the aim is to find the abstract-refinement based abstract representation and evaluation approaches with matching solutions for the verification aspect.

2.2.2 Contributions of this thesis

Brief discussions of our solutions for the three earlier identified key issues related to IAR are addressed.

For the abstract representation aspect, we first specify a given system in XTG (Extended Timed Graph) [5, 56], which is closest to timed automata [3] as a concrete model. After then a new format, PDT (Predicate diagrams for timed systems), which combines XTG and constructs for modelling data, possibly over infinite domain as a way of abstract representation, is introduced.

For the abstract refinement aspect, several proof methods are used in order to guarantee that the result abstract representation is a correct abstraction of a concrete system by conformance checking. Referred to already existing generic abstraction refinement concepts, we relatively detailed our own abstract representation refinement.

For the evaluation aspect, given formulas of the temporal logics (high level properties), and an abstract resentation (PDT) which is obtained by the abstract refinement level, we prove the PDT is the complete-model by checking whethr or not the formula of temporal logic is preserved over the PDT.

We use several tools to support the feasibility of IAR in the fact the involved disciplines are itemized with each technique and tool applied in the brackets below

- modelling systems (XTG; Extended Timed Graph/ PDT; Predicate Diagrams for Timed systems),
- abstraction and refinement (DIXIT; Graphical Toolkit for predicate abstraction/ CVC-LITE; Theorem prover)
- verification (LPMC; Linear Prototype Model Checker or SPIN model checkers/ CVC-LITE; Validity checker)

2.2.3 Relations to other works

A high-level overview of work that is relevant for the work in this thesis is given. In the appropriate chapters, more detailed discussions of related work can be found.

Our abstract representation aspect discussed in this thesis (chapter 3) has its foundation in a broad range of work on predicate diagrams [17] and timed-automata. PDT is a variant of predicate diagram, which is used for the representation and verification of real-timed systems, which are specified in XTG.

Our PDT differs mainly in capabilities of verification for timed systems from the previous predicate diagrams. We added the possibility to represent urgent edges of XTG by giving auxiliary conditions, and extended the use of PDT to verify parameterized systems in the sprit of the work from Baukus et al. [14].

The techniques described in this thesis can be viewed in abstract and refinement sense. Our refinement bears resemblance to refinement as in the B method [16]. It allows one to enrich a model in a step by step approach. Refinement provides a way to construct stronger invariants and also to add details in a model. It is also used to transform an abstract model into a more concrete version by modifying the state description.

Halbwachs [32] successfully applies abstract interpretation to synchronous reactive systems as a way of state space exploration. But he does not consider abstractions over control information (only data information is abstracted). Dill and Wong-Toi [26] use both over- and under-approximations as abstractions, and for finite-state systems, automatically determine whether there are reachable violating sates. Their refinements are different from ours. They refine (over-approximations only) the set of reachable states on paths to violating states. However their techniques are limited to proving invariants.

Predicate abstraction has emerged as a fruitful basis for software verification. Based on predicate abstraction, Namjoshi and Kurshan [51] compute finite bisimulations of timed automata. However, currently it is unclear whether their approach is applicable in practice.

In symbolic model checking for real-time systems, difference bound matrices (DBM) are used to represent a set of state spaces (called regions) over real

variables, and the stereotype of canonical representation seem to be relatively inefficient comparing our abstract representation. Since our abstract refinement based only very limited set of conditions (called conformance checking) and two simple operations (splitting/excluding) that does not justify as much as the effort of maintaining a canonical representation.

Our early work on combining tools for abstract interpretation and state space exploration has been reported in [38]. Although, extra steps used in the algorithm proposed there often fail to significantly reduce the state space, this result identified how to change the direction of that research status towards our generic goal.

Integration of model checking and theorem proving in order to compensate weaknesses of each world and strengthen verification skills that have been described in [4, 25, 52, 27]. In terms of these approaches, our approach combines elements of the latter three by the use of predicate abstraction in the fact that given a concrete infinite state system and a set of abstract predicates, a conservative (but not complete) finite state abstraction is generated. It is conservative (often called conformed) in the sense that for every execution in the concrete system there is a corresponding execution in the abstract system then the abstract version of verification condition is model checked in this abstract system.

The use of predicate diagram as an underlying interface between model checking and theorem proving has been illustrated by Cancell et al. [18, 17, 27]. Although they provided a fine-gained integration of model checking and theorem proving using a mathematically rigorous interface, their approach focused on the verification of non-timed systems

The idea of backward process of IAR is partially inspired by abstract interpretation, which D.Dams used in [24] and mainly influenced by predicate abstraction [25, 52]. Our additional use of theories for abstract refinement purpose (the idea of forward process of IAR) is not found in other work, although there are some similarities with work on interactive theorem proving, which was used to prove correctness of abstraction.

Our approach to decision making temporal properties over abstraction is applied as concerning soundness characterization of abstraction [52] – although there a more efficient solution for the proof of completeness of abstraction is described, her approach is still missing how to establish liveness properties.

In contrast with an executable theory for counterexample guided abstraction refinement (CEGAR) [4, 25, 11, 15, 19] that is in widespread use to find a real bug or to suggest extra predicates to refine the abstraction thereby avoiding that particular spurious trace, our case does not use this CEGAR approach. Instead we more look into the correctness condition of abstraction : at the abstract representation level we consider high-level property as one of abstraction predicates, an abstraction is generated by satisfying the abstraction predicate, in which case the result abstraction is preserving the property. Then we check conformance between an abstraction and a given concrete system as aiming to enrich the abstraction towards being more close to the concrete one. If the abstraction is conformed (pre-

cise enough) to the concrete system and the property is verified then it holds in the concrete system. Otherwise we analyze semi-automatically the failed proof of conformance checking to identify unnecessary (necessary) abstraction predicates to refine the abstraction. Then the process starts anew.

Unlike the CEGAR process which is not guaranteed to terminate, our process can be terminated by proving that two propositions are satisfied: safety property of system is decidable and the abstract is correcly conformed to the system. Also our approach can cope with the verification of liveness property.

2.3 Organization of the thesis

Although part of the material contained in this thesis has been published in the form of articles, it has been restructured and extended. Chapter 3 takes care of the necessary groundwork for specification of given systems such as timed-automata-based formalism XTGs. Also, a common semantic model is defined (timed transition systems), and the formalism is given a meaning in terms of this semantic model.

Chapter 4 studies aspects of abstract representation. Predicate diagrams for timed systems (PDT) is defined as a new format for verification of real-time systems based on which abstract models are generated as a representation of XTGs. Furthermore, the concept of conformance checking is introduced to be able to characterize the result PDT as correct abstract models of XTGs. Also, a concept of structural refinement is presented in order to check the structural conformance between two abstracted models of the same system at different levels of abstraction. Those approaches can be seen as an intermediate step towards IAR presented in chapter 4. Beside, LTL is defined since it is needed for shaping the verification approach and a matching variation of LTL is defined. Finally, methods mentioned can be tested with a small example.

In chapter 5, the work of chapter 3 to chapter 4 is combined resulting in IAR: first the general modelling for real-time systems approach of chapter 3 is used to specify and analyze a given concrete system. Then IAR describes how this concrete system is represented into a complete abstract model (PDT) in terms of two important key aspects; abstract refinement and evaluation, which is discussed in chapter 4.

We consider if an abstract model conforms to its concrete model and if properties of interest is decidable over the abstract model in case both conditions are satisfiable with the abstract model then we can say that the abstract model is a complete abstract representation of its concrete system and the LTL properties hold on the abstract representation also hold on the concrete system. Otherwise, the abstract model is not the complete abstract representation, i.e. the abstract model is not correctly represented or it doesn't preserve the properties. Therefore, we need to iteratively refine the abstract model until it becomes complete and this iterative refinement procedure is shown by one example, Fischer's real

time mutual exclusion protocol.

We also investigate how the abstract representation from IAR leads to the verification framework, the main contribution w.r.t the verification aspect is that the abstract representation built by IAR can be used to prove safety property and livenes property.

Chapter 6 indicates how to extend the proposed work in chapter 5 to the verification of parameterized systems and demonstrated its application at the hand of N-process version of Fischer's real time mutual-exclusion protocol as well.

In chapter 7, we demonstrate the works of chapter 5 and chapter 6 can be applied to not only one specific example (Fischer's protocol) but also some other case study. In fact chapter 6 complements some part of chapter 5 in which the focus was on showing general approaches of IAR, whereis this chapter 7 focuses on additional explaination for the generation of PDTs. Finally, chapter 8 presents conclusions.

Chapter 3

Modelling Real-Time Systems

To be able to perform verification, a formal description of systems and their properties is needed, as well as a common underlying semantic model. This chapter does the essential groundwork of formally defining systems and their properties. It defines a semantic model, a system specification language and a property specification language.

3.1 Basic models

At this stage, we want to make as few assumptions on the data model as possible. A general set of variables is assumed, which can have different types. Only the subset of clock variables has to be made explicit, to be able to conveniently deal with timing aspects. Since a dense time model is used, clocks are real-valued variables.

In the same spirit, value expressions are only defined abstractly. It is only assumed that an evaluation function for value expressions is available. At some point, we will want to be able to require that a value expression is a Boolean value expression. Therefore, the subset of Boolean value expressions is also made explicit.

Definition 3.1 (data language)**.** *A data language provides the following syntactic domains:*

- V *: a finite set of variables,*
- $V_c \subseteq V$ *: a subset of clock variables,*
- $Expr$*: value expressions (over the set* V *of variables), and*
- $Bexpr \subseteq Expr$*: the subset of Boolean expressions.*

We do not fix a precise semantics, but simply require the existence of a suitable semantic domain and evaluation function.

Definition 3.2 (valuation)**.** *We assume a universe Val of values that includes the set $\mathbb{R}^{\geq 0}$ of non-negative real numbers and the Boolean values tt and ff. A valuation is a mapping $\rho : V \to Val$ from variables to values such that $\rho(c) \in \mathbb{R}^{\geq 0}$ for all $c \in V_c$. For a valuation ρ and $\delta \in \mathbb{R}^{\geq 0}$ we write $\rho[+\delta]$ to denote the environment that increases each clock in V_c by δ:*

$$\rho[+\delta](v) = \begin{cases} \rho(v) + \delta & \text{if } v \in V_c \\ \rho(v) & \text{otherwise} \end{cases}$$

We assume given an evaluation function

$$[\![\_]\!]_{\_} : Expr \to (V \to Val) \to Val$$

that associates a value $[\![e]\!]_\rho$ with any expression $e \in Expr$ and valuation ρ. We require that $[\![e]\!]_\rho \in \{tt, ff\}$ for all $e \in Bexpr$.

3.2 Timed transition systems

The underlying model for real-time systems that is employed is that of *timed structures*. In literature other names can be found for similar models. e.g. transition systems and labeled timed transition systems. The key characteristic of a time transition system is that the passing of time is modelled by labelling transitions with nonnegative real numbers. As a consequence, two types of transitions exist: discrete transitions, which have a special label μ and model state changes. Time-passing transitions are labeled by a non-negative real number that represents the amount of time that has elapsed during this transition.

Definition 3.3 (timed transition system)**.** *A timed transition system is a tuple $\langle S, S_0, T\rangle$ where*

- *S is a set of states,*
- *$S_0 \subseteq S$ is the subset of initial states, and*
- *$T \subseteq S \times (\mathbb{R}^{\geq 0} \cup \mu) \times S$ is a transition relation, where μ is a set of labels and the notation $s \xrightarrow{\mu} s'$ is used to indicate a transition $\langle s, s'\rangle \in T$*

such that T has the following properties:

- *For all $s, s', s'' \in S$ and for all $\delta \in \mathbb{R}^{\geq 0}$, if $s \xrightarrow{\delta} s'$ and $s \xrightarrow{\delta} s''$ then $s' = s''$*
- *For all $s, s' \in S$ and for all $\delta, \delta' \in \mathbb{R}^{\geq 0}, s \xrightarrow{\delta+\delta'} s'$, if and only if for some $s'' \in S$, both $s \xrightarrow{\delta} s''$ and $s'' \xrightarrow{\delta'} s'$*

- *For each* $s \in S, s \xrightarrow{0} s$

The three properties included in Definition 3.3 do not play a great role in this thesis, since these will naturally follow from the modelling langauge that generate the semantic models. They mostly concern essential but common-sense qualities of time.

Practically, for formal verification, the notion of runs of a timed transition system is important.

Definition 3.4 (runs of timed transition system)**.** *Given a timed transition system* $\langle S, S_0, T \rangle$*, a* run *of a timed structure is an infinite sequence*

$$\pi = s_0 \xrightarrow{\mu_0} s_1 \xrightarrow{\mu_1} s_2 \ldots$$

where $s_0 \in S_0$ *is an initial state and* $\langle s_i, \mu_i, s_{i+1} \rangle \in T$ *is a transition for all* $i \in \mathbb{N}$*, and* μ_i *is a set of transition labels.*

3.3 Extended Timed Automata Graphs

We model real-time systems as XTGs (extended timed automata graphs) [5, 56] a notation that combines the familiar framework of timed automata [3], synchronous value passing between parallel processes, and a language for modeling data. The semantics of XTGs is defined in terms of timed structures (definition 3.3), also known as timed transition systems.

3.3.1 A brief introduction to XTG

Essentially, an XTG is a finite state machine augmented with clocks and data. A process is modelled by a set of locations (also called control locations) together with a set of edges between these locations. Clocks are non-negative real-valued variables that increase at the same fixed rate (rate one). The increasing of clocks models the progess of time. The execution of transitions of an XTG can be guarded and enforced by constraints on clocks and data. Guards define conditions under which an edge may execute, while location invariants state conditions under which control way resides at that location. The most common use of invariants is to enforce the departure from some location. Upon execution, a transition may update values of clocks and data.

The basic representation for XTG is in the form of a single automaton. Additionally, XTG systems are defined, which allow the specification of multiple automata – possibly representing parallel processes – which synchronized and exchange values. The communication model is based on the synchronous value passing model of value-passing CCS [49]. For example, an edge labelled with a synchronization $l!e$ must synchronize with an edge labelled with $l?v$, where l is a synchronisation label (or channel name), e is a value expression, and v is

a variable. This results in a synchronized transition in which the value of e is assigned to the variable v. In XTG this is generalized to multiple simultaneous value passings, possibly in both directions.

An XTG provides a means for expressing urgency. Edges can be marked as urgent, implying that they should be taken as soon as they are enabled – without letting time pass – indicated by a black dot at an origin location. This general form of urgency allows convenient modelling of edges that trigger on data or time conditions.

Here we present (see figure 3.1 a comprehensive example of an XTG specification. It models two aspects of simple coffee machine, namely an input component that allows a user to request coffee (the left-hand automaton) and the part of the coffee machine that outputs the coffee (the right-hand automaton). c and d are clocks, x and y are integer variables, and p is a parameter.

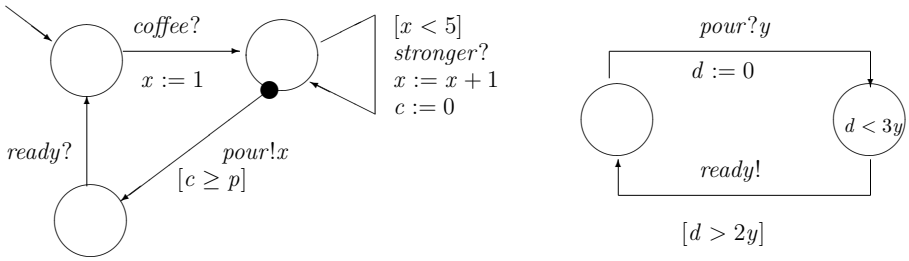


Figure 3.1: *An example XTG system*

The user (which is not shown, the system shown here is incomplete) requests a coffee, by pushing the *coffee* button, which is modelled as a synchronization on *coffee!*. Subsequently it can increase the strength of the coffee by pushing the *stronger* button once or several times, depending on the required strength. Initially the strength – modelled by the real variable x – is equal to 1, while it can at most be 5. Each push on the button increses the strengh with one, until the maximum is reached in which case the button is disabled. (modelled by the guard $[x < 5]$. If for p time-units, the strength was not increased, it triggers the dispenser component to pour the coffee. This is modelled by synchronization on the *pour* channel, passing the strength of the coffee to the dispenser. The clock c keeps track of time, and since the edge is urgent, once p time units have passed , the *pour* command is immediately issued. When synchronizing on *pour*, the dispenser automaton moves to the second location, recording the strength value in y. The second location models the actual pouring of the coffee with strength y, which is not detailed here. The only thing that is known is that the time it takes to pour the coffee is dependent of its strength button, but – due to not modelled factors in pouring the coffee – is not exactly known. This is modelled by an invariant-guard combination, constraining the value of the clock d. Given the strength y, it will take between $2y$ and $3y$ time units to produce the coffee.

Once finishing the delivery of coffee, it signals to the input automaton that it is ready and new cups of coffee can be requested.

3.3.2 XTG systems

This section defines an abstract syntax for XTG. In the description here this is abstracted away by means of the data language model of Definition 3.1. The concrete syntax is however not considered relevant for the discussions in this thesis. The Definition of core syntax and semantics of XTG can be found in [56]. Thus Definition 3.5 presents a subset of full XTG.

Definition 3.5 (XTG)**.** *An XTG process is a tuple* $\langle Init, L, l_0, I, E, U\rangle$ *where*

- $Init \in Bexpr$ *indicates the initial condition for (the data part of) the process,*
- L *is a finite set of locations,*
- $l_0 \in L$ *is the initial location,*
- $I : L \to Bexpr$ *assigns an invariant to each location,*
- $E \subseteq L \times Bexpr \times 2^{V \times Expr} \times L$ *is a set of edges, represented as tuples* $\langle l, g, u, l'\rangle$ *where*
  - $l \in L$ *is the source location,*
  - $g \in Bexpr$ *is a boolean expression, the guard,*
  - $u \subseteq V \times Expr$ *is an update, i.e. a set of assignments, and*
  - $l' \in L$ *is the destination location.*

  Note that an assignment is defined as a set of pairs $\langle v, e\rangle$ *where* v *is a variable and* e *is an expression whose value is to be assigned to the variable. Each variable should appear at most once in the update set.*
- $U \subseteq E$ *identifies the subset of urgent edges.*

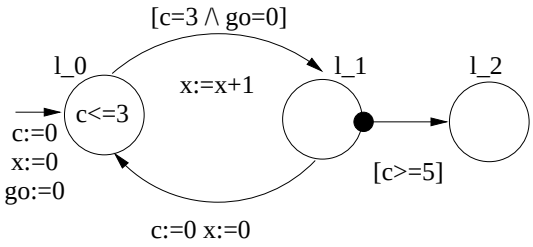
An XTG is a finite set of XTG processes.

Figure 3.2 shows a simple XTG consisting of a single process, both in its a part of textuals (Figure 3.2.a) and graphical (Figure 3.2.b) representations. The XTG process consists of three locations l_0, l_1, and l_2. The edge from l_1 to l_2 is urgent, as indicated by the black dot at the source of the transition in Figure 3.2.b.

Definition 3.6 (configurations of XTG)**.** *A set of configurations of XTG is composed by given a set of variables, constraints, invariants, and conditions of all active locations before and after transitions of XTG systems and it is defined to be* $Conf(l_i)$ *s.t. for all* i, l_i *is the set of active locations of XTG systems.*

```
system example
state integer x:=0,go:=0
process P p1;
graph P
state clock c:=0
init l_0
locations
l_0 inv(c<=3)
{when go=0 and c=3
 do x:=x+1
 goto l_1
}
```

(a) XTG: text form



(b) XTG: graphical form

Figure 3.2: *A single process example of XTG*

An XTG system is formed by the parallel composition of a set of potentially communicating XTG's. Thus, an extended XTG system is defined as a set of XTG's, together with a set of shared variables and a set of communication channels between the individual XTG's. With any XTG we associate a timed structure whose states are given by the active locations of the XTG and the valuations of the underlying variables.

Definition 3.7 (XTG system)**.** *Let $\mathcal{X}$ be an XTG with processes $P_1, \ldots, P_m$. The timed structure $\mathcal{T} = \langle S, S_0, T \rangle$ generated by $\mathcal{X}$ is the structure such that*

- *S_0 consists of all tuples $\langle l_{10}, \ldots, l_{m0}, \rho \rangle$ where l_{i0} is the initial location of process P_i and $[\![Init_i]\!]_\rho = tt$ for the initial conditions $Init_i$ of all processes P_i.*
- *For any state $s = \langle l_1, \ldots, l_m, \rho \rangle \in S$, any $i \in \{1, \ldots, m\}$, and any edge $\langle l_i, g, u, l_i' \rangle \in E_i$ of process P_i such that $[\![g]\!]_\rho = tt$, $\mathcal{T}$ contains a transition $\langle s, \mu, s' \rangle \in T$ where $s' = \langle l_1', \ldots, l_m', \rho' \rangle$ and $l_j' = l_j$ for $j \neq i$, and where*

$$\rho'(v) = \begin{cases} [\![e]\!]_\rho & \text{if } \langle v, e \rangle \in u \\ \rho(v) & \text{otherwise} \end{cases}$$

  provided that $[\![I(l_j')]\!]_{\rho'} = tt$ for all $j \in \{1, \ldots, m\}$.

- *For a state $s = \langle l_1, \ldots, l_m, \rho \rangle \in S$ and $\delta \in \mathbb{R}^{\geq 0}$, $\mathcal{T}$ contains a transition $\langle s, \delta, s' \rangle \in T$ where $s' = \langle l_1, \ldots, l_m, \rho[+\delta] \rangle$ provided that for all $0 \leq \varepsilon \leq \delta$, the location invariants evaluate to true, i.e. $[\![I(l_i)]\!]_{\rho[+\varepsilon]} = tt$, and that for all $0 \leq \varepsilon < \delta$, the guards of any urgent edge $\langle l_i, g, u, l_i' \rangle$ leaving an active location l_i of state s evaluate to false, i.e. $[\![g]\!]_{\rho[+\varepsilon]} = ff$.*

Discrete transitions correspond to edges of one of the XTG processes. They require the guard of the edge to evaluate to true in the source state. The destination state is obtained by activating the target location of the edge and by applying

the updates associated with the edge. Time-passing transitions uniformly update all clock variables; time is not allowed to elapse beyond any value that activates some urgent edge of an XTG process. In either case, the invariants of all active locations have to be maintained.

We still need to extend the XTG towards a concept of communication and synchronisation of values between process. With this intention we introduce the systems of XTG which consist of several XTG communicating through channels of communication, then reduce these systems with basic XTG.

We introduce initially the concept of communication as describing the concept of exchange of values.

Definition 3.8 (communication)**.** *Let $Lab_c = \{lab_{c1}, lab_{c2}, ...\}$ be a set of communication labels among n XTGs, and let $\overline{Lab_c} = \{\overline{lab_{c1}}, \overline{lal_{c2}}, ...\}$ denotes a set of complementary labels. A value exchange expression is a tuple $\langle ch, ia, oa\rangle$ where*

- *$ch \in Lab_c \cup \overline{Lab_c}$ identifies a communication channel,*
- *$ia = \langle v_1, \ldots, v_n\rangle$ is a n-tuple (potentially empty) of variables, and*
- *$oa \in \langle e_1, \ldots, e_n\rangle$ is a n-tuple (possibly empty) expression.*

We indicate that VP_V is an expression of the exchange of values over the set of variables V. Two labels of communication are complementary if one is the overline version of the other, like lab_c and $\overline{lab_c}$.

In concrete syntax, a value passing expression $\langle lab_c, \langle v_1, v_2, ...\rangle, \langle e_1, e2, ...\rangle\rangle$ is wrtten as $lab_c?v_1?v_2...!e_1!e_2, ...$ where $v_1, v_2, ...$ denote variables and $e_1, e_2, ...$ denote value expressions. More often, there is a transfer of only one value, or any value when it is about a pure synchronization. Direction of transfer is indicated by the name of the label. Expressions with complementary labels can synchronize to exchange values, such as for example $\overline{coffee}!3$ and $coffee?strength$ (where *strength* is a variable).

Definition 3.9 (extending XTG systems)**.** *An extending XTG system is a tuple $\mathcal{X} = \langle GInit, G, Ch\rangle$, where*

- *$GInit \in Bexpr$ indicates the global initial condition (variables) for the whole processes,*
- *$G = \langle gr_1, gr_2, ...\rangle$ is a tuple of XTG's*
- *$Ch : EE \to (VP_V \cup \{\perp\})$, where $EE = \bigcup_{\langle Init, I, l_0, I, E, U\rangle \in G} E$,*

such that for all $e, e' \in EE$ with $Ch(e) = \langle lab_c, \langle v_1, ..., v_n\rangle, \langle e_1, ..., e_m\rangle\rangle$ and $Ch(e') = \langle lab'_c, \langle v'_1, ..., v'_n\rangle, \langle e'_1, ..., e'_m\rangle\rangle$, if lab_c and lab'_c are complementary, then $n = m'$ and $m = n'$.

Thus, an XTG system is defined by a global state *GInit*, a set G of single XTG's, and a function *Ch* assigning value passing expressions to some of the edges of the graphs. If $Ch(e) = \perp$ then no value passing is associated with e. The final constraint in the definition only serves to ensure that value expressions with matching labels have maching types. We assume that the identifiers used for locations and local variables are globally unique.

Continuing with semantics for XTG systems, we require to define a synchronization function as follows.

Definition 3.10 (synchronizations)**.** *Let* $vp1 = \langle lab_c, \langle v_1, ..., v_n\rangle, \langle e_1, ..., e_m\rangle\rangle \in VP_V$*, and let* $vp2 = \langle lab'_c, \langle v'_1, ..., v'_n\rangle, \langle e'_1, ..., e'_m\rangle\rangle \in VP_{V'}$*. Then the funtion* $sync(vp1, vp2) \in (2^{(V \cup V') \times Expr} \cup \{\perp\})$ *is defined as follows*

$$sync(vp1, vp2) = \begin{cases} \bigcup_{i \in \{1,...,n\}} \langle v_i, e'_i\rangle \ \cup \bigcup_{j \in \{1,...,m\}} \langle v'_j, e_j\rangle & \text{if } lab_c \text{ and } lab'_c \text{ are complementary} \\ \perp & \text{otherwise} \end{cases}$$

$sync(vp1, vp2)$ returns $\perp$ if $vp1$ and $vp2$ do not communicate, i.e. if the label of synchronization labels are not complementary. If the two value passing expressions match, then an update is produced that is the result of combining the two expressions. (Note: it follows from Definition 3.9, guarantees that $n = m'$ and $m = n'$)

Definition 3.11 (semantics of parallel composition)**.** *Given an extended XTG system* $\mathcal{X} = \langle GInit, \langle gr_1, ..., gr_n\rangle, Ch\rangle$ *with* $gr_i = \langle Init_i, L_i, l_{i0}, I_i, E_i, U_i\rangle$*, the global graph corresponding to* $\mathcal{X}$ *is also an XTG* $\langle Init, L, l_0, I, E, U\rangle$*, where*

- $Init = \bigwedge_{i=1}^{n} Init_i$
- $L = \prod_{i=1}^{n} L_i$
- $l_0 = \langle l_{10}, ..., l_{n0}\rangle$
- $I(\langle l_1, ..., l_n\rangle) = \bigwedge_{i=1}^{n} I_i(l_i)$ *for all* $\langle l_1, ..., l_n\rangle \in L$
- E *and* U *are defined as follows:*
  - *For any edge* $e_i = \langle l_i, g, u, l'_i\rangle \in E_i$ *with* $Ch(e) = \perp$*, the global graph contains edges* $e = \langle\langle l_1, ..., l_n\rangle, g, u, \langle l'_1, ..., l'_n\rangle\rangle$ *for all* $l_j \in L_j$ *(*$j \in \{1, ..., n\} \setminus \{i\}$*), and* $l'_j = l_j$ *for all* j*. Also,* $e \in U$ *iff* $e_i \in U_i$ *for all edges* e*.*

– *If there are two edges* $e_i = \langle l_i, g_i, u_i, l_i' \rangle \in E_i$ *and* $e_j = \langle l_j, g_j, u_j, l_j' \rangle \in E_j$ *for* $i, j \in \{1, ..., n\}$, $i \neq j$ *with* $sync(Ch(e_1), Ch(e_2)) \neq \perp$ *then* E *contains edges* $e = \langle \langle l_1, ..., l_n \rangle, g, u, \langle l_1', ..., l_n' \rangle \rangle$ *where* $g = g_1 \wedge g_2$ *et* $u = u_1 \cup u_2 \cup sync(Ch(e_1), Ch(e_2))$ *for all* $l_k \in L_k$ *(*$k \in \{1, ..., n\} \backslash \{i, j\}$*) and* $l_k' = l_k$ *for all* k. *Also,* $e \in U$ *iff* $e_i \in U_i$ *or* $e_j \in U_j$, *for all edges* e.

The Definition of E and U deserves some explaination. An edge in the global graph originates either from one edge of one of the constituent graphs or – as a consequence of synchronization – from two matching edges from two different graphs. In the first case, the original edge must not have a value passing expression associated with it, since edges with a value passing expression are required to synchronize. The resulting global edge is then given the guard, the update and the urgency attribute from the local edge. In case the edge is the result of synchronization, the two value passing expressions must have matched. Then the guard of the global edge is the conjunction of those of the local edges. The update of the global edge is a combination of the updates of the local edges and the update that results from the synchronization. The global edge is urgent, if either one of the local edge is.

3.4 Conclusion

In this chapter, we defined modelling formalism for model-based specification of real-time systems. Many aspects of what was presented here are based on known results from literature. XTG can be seen as a generalization of timed safety automata [37]. XTG allows the modelling of a generic form of urgency, which proved to be quite useful in specifying verification problems and it is now a concept that is available in many model checking tools [35].

XTG formalism in a general sense allows inclusion any data model that can be described using a denotational semantic model. At the verification perspective level, the data manipulation model brings in an infinite number of states due to unbounded parameters such as time values and standard model checking techniques do not apply to the rich data model. Such limitations on the data model will handle with new notation that we introduce in the next chapter.

Chapter 4

Abstract Representation of XTGs and Evaluation Properties

Besides introducing predicate abstraction and predicate diagram, this chapter presents a systematic abstract representation of given concrete systems. We formalise the notion of abstract representation and a new format of predicate diagrams for verification of real time systems. New results are the characterization of refinement frameworks in terms of conformance checking between abstract representation and concrete systems.

Due to their rich data model, standard real-time model checking techniques do not apply to XTGs. To be able to perform better verification, this chapter illustrates how to abstract (or represent) the behaviour of a given real-time system specified in XTGs. The main idea is to abstract infinite-states data domains into finitely many values so that the discrete control of a given real-time system becomes finite-states and the resulting abstract representation can thus be verified using standard model-checkers.

The verification problem then reduces to (a) establishing the correctness of the abstract representation and (b) evaluating the desired property over the abstract represented model. Because our abstractions give rise to finite-state models, the second subproblem is amenable to model checking.

Concerning the subproblem-(a), we use predicate abstraction and theorem proving techniques. We identify a set of sufficient, non-temporal verification conditions in section 4.5.1. Some basic terms and theorems used throughout the thesis are defined.

Concerning the subproblem-(b), we use model checking technique, however in this section we will handle quite general problem of model checking such that how to establish properties over the result of subproblem-(a). Then a very simple example shortly illustrates how the abstract representation and model checking

work.

4.1 Predicate Abstraction

Predicate abstraction, which is a special form of abstractions, was first presented by Graf and Saidi [53]. In their work, an over-approximation of the reachable states of the abstract system was directly computed. In the work of Colon and Uribe [21], an abstract transition system was computed and then model checked, which is very much close to our approach. The abstract system computed was approximate, to avoid blowing up the number of validity checks needed. But this also means that there was a loss of precision; thus the method could fail to prove properties of the system that could be proved with the most accurate abstract system.

Later work in predicate abstraction [55, 54] made the predicate abstraction method incremental. As new predicates were added to an older abstraction, the new abstract system was derived from the old abstract system. The abstract system was not constructed from scratch each time, thereby avoiding repeating the same proofs.

Predicate abstraction has also been extended to parameterized systems [44]. A counting abstraction was used where the number of processes that satisfied each predicate was tracked.

Comparing above predicate abstraction methods with our approach, we use predicate abstraction as a way of combining theorem proving and model checking for the simple goal as to ease the verification of infinite state systems. An example illustrates how predicate abstraction works w.r.t our goal will be described 4.5.

4.2 Safety properties and Liveness properties

A safety property asserts that a system does not exhibit bad behaviour. An example of a safety property is that the system never goes into an error state. Being able to prove safety properties is desirable, but it is just one facet of system correctness. It is also important to show that the system does something useful, and such properties are called liveness properties. First of all, we will discuss about predicate abstraction according to the safety properties in this chapter. For the liveness properties, we will discuss about it more detail in Chapter 6.

4.3 Predicate Diagram

This section shortly describes a class of diagrams that is used for the verification of discrete system. The objective of this section is to establish better understanding of abstract representation of XTGs, called predicate diagrams for timed systems (PDT) which will be described in section 4.4

Predicate diagrams, first introduced by Cansell et al. [18], is a finite graph whose nodes are labelled with a set of (possibly negated) predicates, and whose edges are labelled with actions (more precisely, action names) as well as optional annotations that assert certain expressions to decrease with respect to an ordering in a finite set of relations. A node of a predicate diagrams represents the set of system states that satisfy the formulas contained in the node. The set and the conjunction of its elements can be written with n. An edge (n, m) is labelled with action A if A can cause a transition from a state represented by n to a state by m. An action may have an associated fairness condition: fairness conditions apply to all transitions labelled by the action rather than to individual edges.

Formally, the definition of predicate diagram is relative to finite set $\mathcal{P}$ and $\mathring{A}$ that contain the state predicates and the (name of) actions of interest. In order to denote the set of literals formed by the predicated in $\mathcal{P}$ a notation $\overline{\mathcal{P}}$ is used as the union of $\mathcal{P}$ and the negations of the predicates in $\mathcal{P}$. This set of predicates contains a finite set $\mathcal{O}$ of binary relation symbols $\prec$ that are interpreted by well-formed orderings. For $\prec \in \mathcal{O}$, the reflexive closure is denoted by $\preceq$. The set of relation symbols $\prec$ and $\preceq$ in $\mathcal{O}$ is denoted by $\mathcal{O}^{=}$.

Definition 4.1 (predicate diagram)**.** *Assume given finite sets $\mathcal{P}$ and $\mathring{A}$ of state predicates and action names. A predicate diagram is given by a tuple $\langle N, N_0, \delta, o, \Omega\rangle$ over $\mathcal{P}$ and $\mathring{A}$ consists of follows:*

- *$N \subseteq 2^{\overline{\mathcal{P}}}$ is a finite set of nodes of the PDT; each node is a pair $\langle l, P\rangle$ for $l \in L$ and $P \subseteq \overline{\mathcal{P}}$,*
- *$N_0 \subseteq N$ is the set of initial nodes,*
- *a family $\delta = (\delta_A)_{A\in\mathcal{A}}$ of relations $\delta_A \subseteq N \times N$; δ is also denoted as the union of the relations δ_A, for $A \in \mathcal{A}$ and we write $\delta_=$ to denote the reflexive closure of the union of these relations,*
- *an edge labelling o that associates a finite set $\{(\eta_1, \prec_1), ..., (\eta_k, \prec_k)\}$, of terms η_i paired with a relation $\prec_i \in \mathcal{O}^{=}$ with every edge $(n, m) \in \delta_A$,*
- *a mapping $\Omega : \mathcal{A} \to \{NF, WF, SF\}$ that associate a fairness condition with every action in $\mathcal{A}$; the possible values represent no fairness, weak fairness, and strong fairness.*

Fairness conditions are used to prevent infinite stuttering. Their interpretation is standard, based on the intuition that the enableness of actions with non-trivial fairness requirement is reflected in the diagram and fairness of an action formula A

- if A is eventually enabled forever (continuously enabled without interrupting), then infinitely many A steps occur

$$WF(A) \equiv \Diamond\Box A \Rightarrow \Box\Diamond A$$

- if A is infinitely often enabled (continually repeatedly enabled), then infinitely many A steps occur

$$SF(A) \equiv \Box\Diamond A \Rightarrow \Box\Diamond A$$

4.4 Predicate Diagrams for Timed Systems

We now describe a variant predicate diagrams, which we call PDT (predicate diagrams for timed systems) [41, 40] that can be used for the verification of real-time systems that differs mainly in capabilities of the verification for timed systems from the previous versions of predicate diagrams in section 4.3.

Essentially, a PDT show a finite-state abstraction of XTGs, and the correctness of the abstraction can be established by proving a number of verification conditions expressed in first-order-logic. In other term, we can say that a PDT is a formalism for representing predicate abstractions of XTGs.

Our interpretation about abstraction (representation) of XTGs can be defined as a special type of a so-called labelled transition system and it is defined in following section.

4.4.1 The PDT Notation

The formal definition of PDTs is given with respect to a set L that represents locations (or, more precisely, location tuples) of the underlying XTG, as well as with respect to a set $\mathcal{P}$ of predicates (i.e., Boolean expressions) of interest. We write $\overline{\mathcal{P}}$ to denote the set containing the predicates in $\mathcal{P}$ and their negations.

Definition 4.2 (PDT and run)**.** *Assume given finite sets L and $\mathcal{P}$. A* PDT *(over L and $\mathcal{P}$) is given by a tuple $\langle N, N_0, R_\mu, R_\tau \rangle$ as follows:*

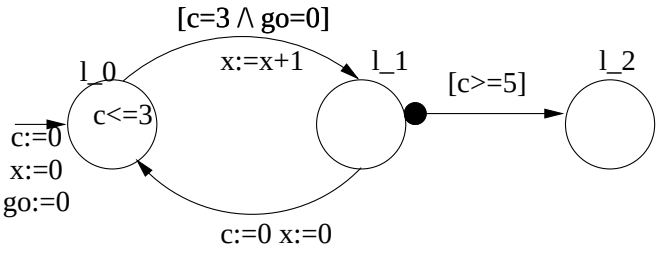
- *$N \subseteq L \times 2^{\overline{\mathcal{P}}}$ is a finite set of nodes of the PDT; each node is a pair $\langle l, P \rangle$ for $l \in L$ and $P \subseteq \overline{\mathcal{P}}$,*
- *$N_0 \subseteq N$ is the set of initial nodes,*
- *$R_\mu, R_\tau \subseteq N \times N$ are two relations that represent discrete and time-passing transitions of the PDT. We require that R_τ be reflexive. We usually write $n \rightarrow_\mu n'$ and $n \rightarrow_\tau n'$ for $(n, n') \in R_\mu$ and $(n, n') \in R_\tau$.*

A run (path) *of a PDT is an infinite sequence*

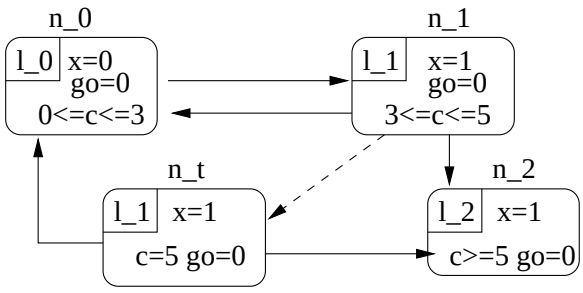
$$\sigma \quad = \quad n_0 \xrightarrow{lab_0} n_1 \xrightarrow{lab_1} n_2 \ldots$$

where $n_0 \in N_0$, $lab_i \in \{\mu, \tau\}$, and $n_i \rightarrow_{lab_i} n_{i+1}$ for all $i \in \mathbb{N}$.

Thus, a PDT is a labelled transition system with two transition relations. A PDT node represents a set of XTG states by indicating the active locations and certain predicates satisfied by these states. Figure 4.1.(b) shows a PDT represented from an XTG in Figure 4.1.(a). The transition relations correspond to discrete transitions and time-passing transitions of the XTG.



(a) XTG: graphical form



(b) A PDT for this XTG

Figure 4.1: *An abstract representation example*

When drawing a PDT, as in Figure 4.1.(b), we use solid arrows for edges in R_μ and dashed arrows for edges in R_τ. Every node has a τ-loop associated with it, which we do not show explicitly.

Several notions are defined to work on the predicates defined over concrete systems that lead to important terms in the remainder of the thesis.

Notice that in the abstract representation, the concrete system XTG is described as a list of its configurations of each location. Each configuration consists of a boolean expression called a set of invariants, guards and action statements that modifies the XTG locations.

A set of abstraction-predicates is a set of predicates that is expressive enough to do abstract representation of XTG systems.

Definition 4.3 (abstraction-predicates)**.** *Let $Conf(\mathcal{X})$ be a set of configurations of XTG systems, XTG and let Ψ be a set of predicates $\mathcal{P}$. Then a set*

of abstraction-predicates Ψ *with respect to* $Conf(\mathcal{X})$ *is any formula with the set of free variables in* $Conf(\mathcal{X})$

A set of abstraction-predicates Ψ determines an abstract representation function α, which maps each evaluation of each location to a bitvector b of length of the number of elements of Ψ if and only if the i-th element of Ψ holds for the evaluation.

4.5 Abstract Representation

In this section, we describe the abstract representation of XTGs using a PDT. In order to show completeness of abstraction, i.e. – for any specification and any formula of the temporal propositional logic, if the specification (in which case XTGs) implies the formula, then the implication can be proven by a suitable predicate diagram – PDT should satisfy two conditions: first, all behaviour allowed by formal specification of a given system such as XTG must also be traced through the abstract representation such as PDT. Second, every run (path) through the PDT must satisfy property F.

To precise that PDT is a correct abstract representation of XTG, we consider labels in nodes of PDT as abstraction-predicates on the concrete state space of XTG, and reduce run inclusion to a set of proof obligations that concern individual states and transitions.

On the other hand, to show that the PDT implies the high level properties (temporal properties), we also regard all labels as Boolean variables. The PDT can therefore be encoded as finite labelled transition system, whose temporal properties are established by model checking. We now consider the first aspect, abstract representation, in the following sections.

4.5.1 Conformance: Relating XTG and PDT

We now formally define what it means for a PDT to conform to an XTG, i.e. when the PDT is a correct abstract representation of the XTG. A PDT Δ is said to conform to an XTG $\mathcal{X}$, written $\mathcal{X} \lhd \Delta$, if every behaviour that satisfies $\mathcal{X}$ is a run through Δ. We also establish a set of verification conditions that guarantee conformance. Our purpose in defining conformance is to ensure that any property verified over the PDT also holds for the XTG. Because we are interested in verifying linear-time properties, and such properties hold of a system if they are satisfied by each system run, we should verify that each run of an XTG can be mapped to a run of the PDT. The following definition makes this intuition precise.

Definition 4.4 (trace)**.** *Given an XTG* $\mathcal{X}$*, a PDT* Δ*, and a run* $\pi = \langle l_0, \rho_0 \rangle \xrightarrow{\lambda_0} \langle l_1, \rho_1 \rangle \ldots$ *of* $\mathcal{X}$*, we say that a run* $\sigma = n_0 \xrightarrow{lab_0} n_1 \ldots$ *of* Δ *is a* trace *of* π*, is denoted by* $\sigma = tr(\pi)$*, iff*

- *π and σ are of equal length (both are infinite),*
- *$n_i = \langle l_i, P_i \rangle$ for some $P_i \subseteq \overline{\mathcal{P}}$ such that $[\![p]\!]_{\rho_i} = tt$ for all $p \in P_i$ and all i, i.e. the states of π and the nodes of σ activate the same locations and all predicates of n_i are satisfied in the corresponding state of π, and*
- *$lab_i = \mu$ if $\lambda_i = \mu$, and $lab_i = \tau$ if $\lambda_i \in \mathbb{R}^{\geq 0}$, i.e. the two runs agree on which transitions are discrete and which are time-passing.*

We say that Δ conforms to $\mathcal{X}$ if every run of $\mathcal{X}$ has a trace in Δ.

The definition of conformance requires to inspect all runs of an XTG. For practical purposes, we are interested in establishing a reasonably small set of first-order verification conditions that are sufficient to ensure conformance. The following theorem gives such conditions. Intuitively, we verify that every possible initial state of the XTG is represented by some initial node of PDT. Inductively, given any XTG state s corresponding to some PDT node n and any transition from s to some successor XTG state s', that transition can be mapped to a transition from node n in the PDT. In formulating the verification conditions, we introduce two copies V' and V'' of the set of variables V whose elements are decorated with single and double primes (v' and v'' for each $v \in V$). When P is a set of predicates, we sometimes also denote by P the conjunction of the predicates in P, and we write P' or P'' to denote the formula obtained by replacing each variable $v \in V$ by its copy v' or v''.

Theorem 4.5. *Let $\mathcal{X}$ an XTG that consists of m processes $P_i = \langle Init_i, L_i, l_{i0}, I_i, E_i, U_i \rangle$, and that $\Delta = \langle N, N_0, R_\mu, R_\tau \rangle$ is a PDT over $L_1 \times \cdots \times L_m$ and a set $\mathcal{P}$ of predicates. If all of the following conditions hold then Δ conforms to $\mathcal{X}$:*

1. $$\bigwedge_{j=1}^{m} Init_j \wedge I(l_{j0}) \quad \Rightarrow \quad \bigvee_{\langle l_{10},\ldots,l_{m0},P\rangle \in N_0} P$$

   In words, the conjunction of the initial conditions of $\mathcal{X}$ and the invariants of the initial locations imply that the predicates of one of the initial nodes of Δ marked with the initial locations must be true.

2. *For any node $n = \langle l_1, \ldots, l_m, P \rangle$ of Δ and any edge $\langle l_i, g, u, l_i' \rangle$ of XTG process P_i, let V_u denote the set of variables v that are updated by u (i.e. such that $\langle v, e \rangle \in u$ for some e), and let N' denote the set of all nodes $n' = \langle l_1', \ldots, l_m', Q \rangle$ where $l_j' = l_j$ for $j \neq i$ such that $n \rightarrow_\mu n'$.*

   $$P \wedge g \wedge \bigwedge_{j=1}^{m} (I(l_j) \wedge I'(l_j')) \wedge \bigwedge_{\langle v,e\rangle \in u} v' = e \wedge \bigwedge_{v \in V \setminus V_u} v' = v \quad \Rightarrow \quad \bigvee_{\langle l_1',\ldots,l_m',Q\rangle \in N'} Q'$$

   In words, the predicate label of node n and the invariants of all active locations before and after the transition of $\mathcal{X}$ should imply the predicate label of some node in N'.

3. *For any node $n = \langle l_1, \ldots, l_m, P\rangle$ of Δ, let N'' denote the set of all nodes $n'' = \langle l_1, \ldots, l_m, Q\rangle$ that agree with n on the location components such that $n \rightarrow_\tau n''$.*

$$P \wedge \delta \in \mathbb{R}^{\geq 0} \wedge \bigwedge_{c \in V_c} c' = c + \delta \wedge \bigwedge_{v \in V \setminus V_c} v' = v \wedge \bigwedge_{j=1}^{m} I(l_j) \wedge I'(l_j)$$
$$\wedge\ \forall \varepsilon \leq \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \ \Rightarrow \bigwedge_{j=1}^{m} I''(l_j)$$
$$\wedge\ \forall \varepsilon < \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \ \Rightarrow \bigwedge_{j=1}^{m} \bigwedge_{\langle l_j, g, u, l_j'\rangle \in U_j} \neg g''$$
$$\Rightarrow \bigvee_{\langle l_1, \ldots, l_m, Q\rangle \in N''} Q'$$

In words, assuming the predicate label of n and the invariants of all active locations before and after a time passing transition by amount δ that does not activate any urgent transition of $\mathcal{X}$, the PDT must contain some node n'' that is reachable from n by a τ-transition and whose predicate label is guaranteed to hold.

Proof. Given a run $\pi = \langle l_0, \rho_0\rangle \xrightarrow{\lambda_0} \langle l_1, \rho_1\rangle \ldots$ of $\mathcal{X}$, we can inductively construct a trace σ of π in PDT Δ as follows: because ρ_0 must satisfy the initial conditions of all processes as well as the invariants of the initial locations, condition 1 ensures that there exists some initial node of Δ that is associated with the tuple of initial locations of $\mathcal{X}$ and whose predicate label is true in ρ_0. Inductively, assume that a node $n = \langle l, P\rangle$ corresponding to the XTG configuration $s_i = \langle l_i, \rho_i\rangle$ has already been identified. If the transition in π from s_i is a discrete transition, it is due to some edge of some process P_j (cf. Definition 3.7 condition 2), and therefore the guard of that edge must be true in ρ_i and its updates will be performed during the transition to state $\langle l_{i+1}, \rho_{i+1}\rangle$. Moreover, the location invariants must be true in the states before and after the transition. According to condition 2) we can therefore find a node n' associated with l_{i+1} such that $n \rightarrow_\mu n'$ and that the predicate label of n' holds in ρ_{i+1}.

Similarly, if the transition in π from s_i is a time-passing transition, it is due to some δ-labelled time edge of some process P_i (cf. Definition 3.7 condition 3), the valuation of the destination edges properly the passing of δ time units must be true in $\rho_i[+\delta]$, and its update will be performed during the time passing transition to state $\langle l_i, \rho_i[+\delta]\rangle$. While increasing the clocks with δ, the location invariants must be true in the states before and after the time passing transition and do not have enabled urgent transition of the location. According to condition 3 we can therefore find a node n'' associated with l_i such that $n \rightarrow_\tau n''$ and that the predicate label of n'' holds in $\rho_i[+\delta]$. □

The above set of conditions of Theorem 4.5 guarantees that the result abstraction (PDT) can capture all runs of the given system (XTG). These conformance

conditions can be formulated with first-order logic and verified by theorem provers.

4.5.2 Structural refinement

Beyond their use in checking systematic conformance between PDT and XTG, PDT can also be used to compare two abstract models of the same system at two different levels of abstraction. For instance, Figure 4.2.(a) – (c) shows three abstract models at different levels of abstraction for the same system in Figure 4.1.(a).

We write $\Delta^i \preceq \Delta^{i+1}$ ("Δ^1 refines Δ^2 ") if any execution of the PDT Δ^1 is also one execution of Δ^2. It is supposed that the set of predicates underlying Δ^1 extends the corresponding set underlying Δ^2. For the proof of conformance, we are interested in "local" conditions in order to establish refinement between two diagrams, and we define the concept of structural refinement as follows.

Definition 4.6 (structural refinement)**.** *Let $\Delta^i = \langle N^i, {N_0}^i, {R_\mu}^i, {R_\tau}^i \rangle$ (for $i = 1, 2$) two predicate diagrams over L and $\mathcal{P}$, and $f : N^1 \to N^2$ is a function. The PDT Δ^1 is a structural refinement of Δ^2 w.r.t. f if the following conditions are verified:*

1. *$\models n \Rightarrow f(n)$ for every node $n \in N^1$,*
2. *$f(n) \in {N_0}^2$ for all $n \in {N_0}^1$,*
3. *for all $n, n' \in N^1$ such that $(n, n') \in R^1_\varrho$ (where $\varrho \in \{\mu, \tau\}$) we have $(f(n), f(n')) \in (R^2_{\mu\cup\tau})^=$, the reflex closure of the union of relations of transition in Δ^2.*

We say that Δ^1 is a structural refinement of Δ^2 w.r.t some function $f : N^1 \to N^2$.

The conditions 1 – 3 ensure that there is a strong correspondence between the transition graphs from the two different diagrams: node labels of Δ^1 imply those of the corresponding nodes of Δ^2, initial nodes of Δ^1 correspond to the initial nodes of Δ^2, and the transitions in Δ^1 must map to transitions in Δ^2.

The following Theorem 4.7 asserts that structural refinement ensures execution inclusion, and is therefore sound. In particular, all properties and predicates shown of Δ^2 remain valid for Δ^1.

Theorem 4.7. *If Δ^1 is a structural refinement of Δ^2 w.r.t the function f, then for any execution $n_0 \xrightarrow{lab_0} n_1 \xrightarrow{lab_1} \ldots$ of Δ_1, the sequence $(f(n_0), f(n_1), \ldots)$ is an execution in Δ^2 which can contain the excutions of nodes.*

Proof. Assume that Δ^1, Δ^2 and f are as in Def 4.6, we must prove an execution $\sigma = n_0 \xrightarrow{lab_0} n_1 \xrightarrow{lab_1} n_2 \ldots$ of Δ^1 is an execution of Δ^2.

We will define a trace $\pi = f(n_0) \xrightarrow{lab_0} f(n_1) \xrightarrow{lab_1} f(n_2) \ldots$ of Δ^1 and prove π is an execution of Δ^2. Define the sets $\mathit{ff}(lab_i)$ by

$$\mathit{ff}_i = \{\varrho \in {R_{\mu\cup\tau}}^2 : (f(n), f(n')) \in {R_\varrho}^2\}$$

and let $ff_i{}^* = ff_i$ if $f(n_i) \neq f(n_i')$, and $ff_i = \tau$ otherwise. Condition 3 of Definition 4.6 ensures that $ff_i{}^* \neq 0$ holds for all $i \in \mathbb{N}$. Now choose a sequence $f(n_0), f(n_1), ...$, of relations $f(n_i) \in ff_i{}^*$ such that every relation $\varrho \in R_{\mu\cup\tau}{}^2$ that appears in infinitely many sets $ff_i{}^*$ is chosen infinitely often. (such a choice is possible because the set $R_{\mu\cup\tau}{}^2$ is finite.)

Since σ is an execution of Δ^1, we know that $n_0 \in N_0{}^1$, hence condition 2 of Definition 4.6 ensures that $f(n_0) \in N_0{}^2$. Inductively, assume that a node n_i corresponding to the node $f(n_i)$ has already been identified, and condition 1 implies $n_i \models f(n_i)$.

Similarly, the choice of $f(n_i)$ and condition 3 ensures that either $f(lab_i) = \tau$ and $f(n_i) = f(n_i')$ or $f(n_i) \in R_{\mu\cup\tau}{}^2$ and $(f(n), f(n')) \in R_{f(n_i)}{}^2$ □

4.6 Model checking: evaluation properties

Assume that we have a Δ such that $\mathcal{X} \lhd \Delta$, now we turn to evaluating the desired properties of $\mathcal{X}$ over the Δ. We assume that the properties of interest are expressed in linear-time temporal logic LTL, and that they are built from the predicates in $\mathcal{P}$. We can thus simply consider the predicates that appear as labels of the Δ as uninterpreted atomic propositions.

Viewing a Δ as a finite-state transition system, temporal properties of its traces can be established using LTL model checking. This Δ can be encoded in the modelling language of conventional finite-state model checkers. For our experiments, we use either the Spin model checker via the DIXIT toolkit [27] or PMC [12] model checker for manipulating and analyzing LTL formulas over Δ.

4.6.1 Linear Temporal Logic

Our property specification language LTL [29] allows assertions about temporal behaviour of a system. Given a finite set of atomic propositions AP, the LTL formulas are defined inductively as follows:

- every member of AP is a formula
- if φ and ψ are formulas, then so are $\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi, \bigcirc\varphi$, and $\varphi \,\mathrm{U}\, \psi$

A ω-word $\zeta = x_0, x_1, ...$ over the alphabet 2^{AP}, i.e. a mapping from the naturals to 2^{AP}. We write ζ^i for the suffix of ζ starting at x_i. The semantics of LTL is as follows:

- $\zeta \models A$ iff $A \in x_0$, for all $A \in AP$,
- $\zeta \models \neg\varphi$ iff $\zeta \not\models \varphi$,
- $\zeta \models \varphi \wedge \psi$ iff $\zeta \models \varphi$ and $\zeta \models \psi$,
- $\zeta \models \varphi \vee \psi$ iff $\zeta \models \varphi$ or $\zeta \models \psi$,

- $\zeta \models \circ\varphi$ iff $\zeta^1 \models \varphi$,
- $\zeta \models \varphi \text{ U } \psi$ iff there is an $i \geq 0$ such that $\zeta^i \models \psi$ and $\zeta^j \models \varphi$ for all $0 \leq j < i$.

Let F be an abbreviation for $A \wedge \neg A$, and T be an abbreviation for $\neg\mathsf{F}$. We also use the following abbreviations: $\varphi \vee \psi = \neg((\neg\varphi) \wedge (\neg\psi))$, $\Diamond\varphi = \mathsf{T} \text{ U } \varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$, and $(\varphi \Rightarrow \psi) = (\neg\varphi \vee \psi)$.

In general, the soundness of verification methods involving abstractions typically depends on the following property: whenever the abstract system satisfies a formula, then the concrete system also satisfies this formula. Since LTL formulas are interpreted over all possible behaviours, it suffices to show that every behaviour of the concrete system has a corresponding behaviour in the abstract system.

Theorem 4.8 (Soundness of PDT)**.** *For LTL formula* φ, $\Delta \models \varphi \Rightarrow \mathcal{X} \models \varphi$

Proof. Assume that Δ conforms to $\mathcal{X}$. The theorem can be proved by structural induction over the formula and it is based on the fact that the definition of conformance and Theorem 4.5 and thus ensures that for every trace in $\mathcal{X}$ a "corresponding" trace in Δ can be found. It follows that any LTL property φ built from predicates in $\mathcal{P}$ that holds over some Δ also holds of $\mathcal{X}$. Indeed, let π be any run of $\mathcal{X}$. Because Δ conforms to $\mathcal{X}$, we can find a trace σ of π in Δ. Since φ is assumed to hold of Δ, it follows that σ satisfies φ, and given that only predicates in $\mathcal{P}$ appear in φ, a straightforward induction on LTL formulas shows that φ must also hold of π. □

By the soundness of Δ in Theorem 4.8, we can guarantee that if Δ satisfies a given property φ then $\mathcal{X}$ also satisfies φ.

On the other hand, if a property fails in Δ, abstract counterexamples trace are produced by the model checker and we try to find a corresponding concrete counterexample trace. In fact, the abstract counterexample trace needs not correspond to actual one because some detail may have been lost in the abstraction. Such tracing up the abstract counterexamples can be helpful for refining the abstraction and this methods are very well known and have been widely used in many other researches [28, 15, 11, 19].

Unlike diagnosing conterexamples for the refinement, our approach aims to prevent causing false negative result: given a concrete system and abstraction predicates, we construct an abstract representation, which preserves the property φ. Thus the abstract representation satisfies φ at any abstract representation level although the abstract representation is not correctly conformed to the given system. Therefore we need to refine this abstract representation as applying conformance checking again. We will explain details of this approach in next chapter.

4.7 An example

This section gives a taste of what the predicate abstract representation of XTGs looks like. For a very small XTG-PDT representation problem, we recall that an example 3.2 in section 3.3.2, which is also shown in Figure 4.1.(a), and we give a little taste of our abstract representation refinement and more details will be discussed in chapter 5.

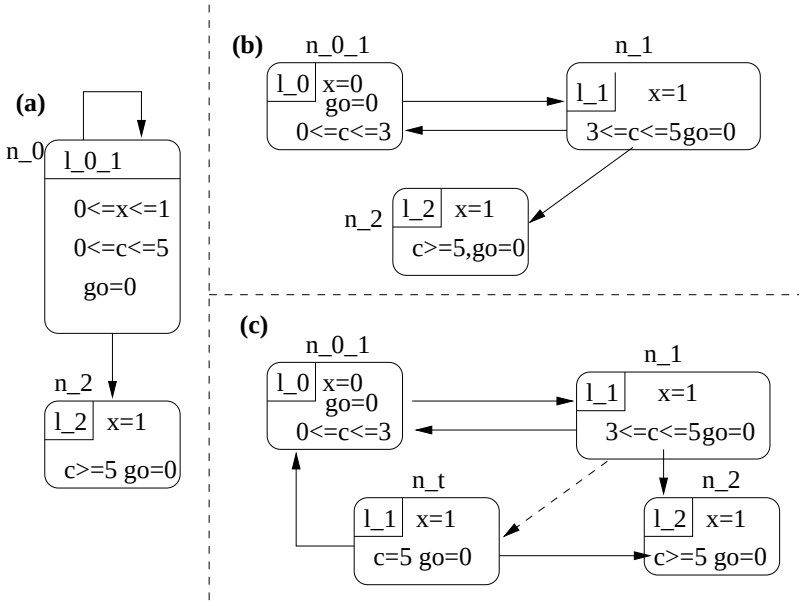


Figure 4.2: *An example of predicate abstract-representation*

Figure 4.2 shows different possible abstract representations of the XTG systems. Also those stages (a), (b) and (c) can be the sequences of operations that would occur when dealing with the abstract representation refinement. It shows three snapshots of three possible abstract representations of a same system at different abstraction level. In Figure 4.2, a node is implicitly interpreted as a region in a sense that a region is the set of locations and its configuration of XTG.

We then have an addressed question like "is it necessary to go through this refinement stages (or sequences of operations)?" The answer is to be either ' 'yes" or "no" depends on cases what our goal would be. If the goal is just limited to show that the given system is possibly represented as a PDT then the operation can be stop at any stage. In other cases such that the goal is to decide properties hold on the abstract representation and to guarantee this abstract representation conforms to XTG then such a going-through-refinement-step is inevitable in order to reach the final stage (c) and we say that the abstract representation at (c) is

the complete model we are searching for.

Suppose that the PDT shown in Figure 4.2.(a) is a first abstraction of the given XTG: XTG consists of a set of its configurations. We use a set of abstraction predicates $\Psi = \{0 \leq x \leq 1, 0 \leq c \leq 5, go = 0, c \geq 5\}$ to construct the first abstract representation. The predicates are just possible rages of given configurations of XTG system.

However, this abstract representation (a) is not the ideal one in order to decide if a property holds on it since (a) is constructed under over-approximation, which means that there was a loss precision: a PDT might not contain the atomic formulas that occur in the property. To precise more about such a deciding property problem, we consider a property we like to check first,

$$\Box(x = 1 \Rightarrow c \geq 3) \tag{4.1}$$

then we see that we cannot guarantee that a node n_0 is decidable for the given property 4.1 since there exist cases that the property is satisfied or not satisfied.

For example in a case that the node n_0 has predicates x=1, c=0 and go=0 then the property cannot be held in this node but except that case this property are held. Thus, we don't know (cannot decide) clearly about whether or not this property is held.

To solve the decidability problem (in order to refine the first abstraction), now we consider building a second abstract representation-(b) in Figure 4.2 as checking two propositions: first, the second abstract representation-(b), which mapped to the first abstract representation (a), is constructed by checking the structural refinement proposition between two different abstraction representations in the sense that every node of the diagram has been derived from nodes of the higher-level diagram and that relations of the lower level respect the relations that existed before.

More specifically, conditions 2 and 3 of Definition 4.6 can be verified by inspection of the graph structure with the help of DIXIT [27], which is a graphical editor for the specification, refinement, and verification of reactive systems, based on the concept of predicate diagrams representing Boolean abstractions. The full concept/techiques of DIXIT is however not considered relevant for the discussions in this thesis because of some limitation of current DIXIT such that the condition 1 of Definition 4.6 requires (non-temporal) theorem proving, however currently this condition cannot be verified by DIXIT.

For that reason, in terms of our three key aspects we partially use the DIXIT toolkit for drawing a predicate diagram including its refined one at the different abstract representation levels and verifying given properties at the evaluation level.

Now, we consider the second proposition, i.e. systematic conformance checking as comparing between an abstract representation-(a) and its relate XTG in order to handle the limitation of current DIXIT toolkit. A set of proof-obligations can be discharged based on the three conditions of Theorem 4.5 and the set of

proof-obligations can be proven by SMT-solvers such as CVC-LITE [23], which are fully automated theorem provers (or validity checkers) for combinations of certain theories, including linear arithmetic and some support for quantifiers.

As doing the refinement of the first abstract representation-(a), the node n_0 in Figure 4.2.(a) is split into n_{01} and n_1 in Figure 4.2.(b) w.r.t $Conf(l_0)$ and $Conf(l_1)$ in Figure 4.1.(a). Each node has labels satisfying the corresponding configurations and the node has τ-loop associated with it implicitly. Now we see if the property (4.1) is decidable in every node of the abstract representation-(b) in Figure 4.2.(b)

In fact the location l_1 of XTG has an urgent transition, which means that this current abstract representation-(b) could have a certain node that describes the urgent condition explicitly. For the reason, we add a new node n_t, which is reachable from n_1 by a τ-transition. But the system cannot stay in both of n_1 and n_t more than time-unit 5 ($c > 5$). The result abstract representation is depicted in Figure 4.2.(c), which is actually the same PDT in Figure 4.1.(b).

The last step for this example is now to prove that the result abstract representation in Figure 4.2.(c) conforms to the XTG of Figure 4.1.(a) by doing systematical conformance checking.

For example, for the initial condition of Theorem 4.5, we obtain the proof obligation

$$c = 0 \wedge x = 0 \wedge go = 0 \wedge c \leq 3 \quad \Rightarrow \quad c \leq 3 \wedge x = 0 \wedge go = 0$$

As an example for the verification conditions of type 2 of Theorem 4.5, we consider the XTG transition from l_0 to l_1, which has to be matched with the transitions leaving node n_0 of the PDT:

$$\begin{array}{ll} & x = 0 \wedge c \leq 3 \wedge go = 0 \wedge c = 3 \wedge go = 0 \wedge x' = x + 1 \wedge go' = go \wedge c' = c \\ \Rightarrow & x' = 1 \wedge 3 \leq c' \wedge c' \leq 5 \wedge go' = 0 \end{array}$$

Finally, we consider the possible time passing transitions leaving location l_1, focussing on the PDT node n_1:

$$\begin{array}{ll} & x = 1 \wedge 3 \leq c \wedge c \leq 5 \wedge go = 0 \wedge \delta \in \mathbb{R}^{\geq 0} \wedge c' = c + \delta \wedge x' = x \\ & \wedge go' = go \wedge \ \forall \varepsilon < \delta : c'' = c + \varepsilon \wedge x'' = x \wedge go'' = go \Rightarrow \neg(c'' \geq 5) \\ \Rightarrow & (x' = 1 \wedge 3 \leq c' \wedge c' \leq 5 \wedge go' = 0) \vee (x' = 1 \wedge c' = 5 \wedge go' = 0) \end{array}$$

Observe in particular that time cannot advance beyond a clock value of 5 because the transition from l_1 to l_2 is marked as urgent. We thus, arrive at a conventional concrete abstract representation of the given small XTG example, as shown in Figure 4.2.(c).

4.8 Conclusion

This chapter defined a new format of predicate diagrams(Δ), which are succinct and intuitive representations of Boolean abstractions, for the verification of real-

time systems. It advocated the use of predicate diagrams for abstracting the behavior of a given real-time system specified extended timed automaton graphs.

We also defined the LTL property, as well as formal models that enable formal reasoning over the Δ, and explained very shortly how to construct Δ, and how to evaluate the LTL property over it

The main contribution of this chapter is that we established a set of verification conditions, Theorem 4.5, that are sufficient to prove that a given Δ is a correct abstract representation of $\mathcal{X}$. We also showed that Δ can also be used to compare two abstract models of the same system at different levels of abstract representation and such a structural refinement is also quite handy for the bottom-to-top refinement approach from the very beginning of the first over-approximation.

Although applying our methodology to the more complex system was not really completely elaborated in this chapter, the major contribution made here results in obtaining the valid abstract model and identifying necessary refinements so far.

The next question addressed now is how to find the required predicates as identifying necessary refinements and how to verify the desired property with more complex systems, and the answers will be presented in next chapters.

Chapter 5

Iterative Abstract Refinement

In previous chapter 4, abstract representation and its evaluation with LTL formula have been introduced. Also a related example has been presented. This chapter continues going into the characterization of the abstract representation in terms of introducing IAR (Iterative Abstract Refinement). IAR is the method of combining predicate abstraction, theorem proving, and model checking techniques that constructs a complete abstract representation.

In this chapter, we also investigate how the PDT Δ resulted from IAR leads to the verification framework, which shows that the system does not exhibit bad behaviour and the system does something useful.

Assume given a real-time specification in XTG $\mathcal{X}$ and a property $\mathcal{F}$. We recall that in model checking perspective formalism, the proof that $\mathcal{X}$ satisfies $\mathcal{F}$ can be considered as proving the validity of $\mathcal{X} \models \mathcal{F}$. Following the approach of the verification of IAR using Δ, we do the proof with both: – proving correctness of Δ as finding Δ such that every model of $\mathcal{X}$ is a trace through Δ, and – proving that every trace through Δ is a model of $\mathcal{F}$.

Proving the correctness property is done by considering nodes' labels in Δ as abstraction predicates on concrete state space of $\mathcal{X}$ and reducing the trace inclusions to a set of first-order verification conditions that concern individual states and transitions. Thus, this one method is done deductively. On the other hand, the other method will be done by regarding the node labels related to $\mathcal{F}$, and probably the auxiliary conditions, as invariants (Boolean variables) and then encode Δ as a finite labelled transition system. The $\mathcal{F}$ then can be established by model checking over Δ.

One of the contributions of this thesis is that our abstract representation format Δ can be used in order to prove not only safety property but also liveness property. This chapter describes how safety and liveness of Δ are proven. Not surprisingly, liveness will turn out a lot harder to prove than safety. We will explain what the auxiliary conditions are, and define several new terms that can

be used to verify liveness property over Δ.

We first gives the necessary background of this chapter in section 5.1 then we introduce the idea of IAR on an intuitive level in section 5.2. As illustration, we present the verification of Fischer's protocol as an example in the rest of this chapter.

5.1 Predicate Abstraction

The relationship between concrete and abstract states that underlies abstract interpretation is traditionally described by Galois connection, which is defined below. We use the concept of Galois connection in the construction of predicate diagram as well.

Definition 5.1 (Galois connection)**.** *Let* $(L_1, \sqsubseteq_1)$*, and* $(L_2, \sqsubseteq_2)$ *be partially ordered sets (posets) and let* $\alpha : L_1 \to L_2$ *and* $\gamma : L_2 \to L_1$ *be functions. The pair* (α, γ) *is said to form a Galois connection if*

$$\forall x \in L_1, y \in L_2 : x \sqsubseteq_1 \gamma(y) \leftrightarrow y \sqsubseteq_2 \alpha(x)$$

The set L_1 *and* L_2 *are called the concrete and abstract domain, respectively. The function* α *is called the abstraction function and* γ *is called the concretization function*

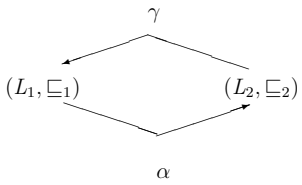


Figure 5.1: *Galois connection between* $(L_1, \sqsubseteq_1)$ *and* $(L_2, \sqsubseteq_2)$

Graphically, we can denote a Galois connection, for example, as is shown in Figure 5.1. In our setting, we choose the set of configurations of XTG, $Conf(\mathcal{X}) = \{Conf(l_1), Conf(l_2), \ldots\}$, as L_1 and a set of nodes of PDT, N, as L_2. The abstraction function returns a set of nodes such that

$$\alpha(Conf(\mathcal{X})) = \{n \in N : \forall Conf(l_i) \in Conf(\mathcal{X}) : Conf(l_i) \in \gamma(n)\}$$

and the concretization function γ, produces a set of configurations of locations of XTG that are models of a node, is defined by

$$\gamma(n) = \{Conf(l_i) \in Conf(\mathcal{X}) : Conf(l_i) \models n\}$$

Similarly, in our setting a set of configurations for each location i can be also expressed in a tuple $\langle l_i, \rho_i \rangle$ and the value of an abstraction-predicate ψ w.r.t. a

valuation ρ from a tuple of location is denoted by the juxtaposition $\psi\rho$. Whenever $\psi\rho$ evaluates to *true*, we write $\rho \Vdash \psi$

A set of abstraction-predicates $\Psi = \{\psi_1, ..., \psi_n\}$ determines an abstraction function α, which maps each valuation ρ to a bit-vector b of length of n such that the i-th component of b is *true* if and only if ρ holds for the i-th element of Ψ. Here, we assume that bit vectors of length n are elements of the set B_n, which are functions of domain $\{1, ..., n\}$ and codomain $\{0, 1\}$. The inverse image of α, that is, the concretization function γ, maps a bit-vector to the set of valuations which satisfy ψ_i. Thus, a set of configurations of concrete locations $\langle l, \rho\rangle$ is transformed by the abstract representation function α into the abstract node $\alpha(\langle l, \rho\rangle)$, and an abstract node is mapped by γ to a set of concrete locations $\gamma(\langle l, b\rangle)$.

The α and γ can be defined as concerning our setting. These extended definitions are as follows:

Definition 5.2 (Abstraction/Concretization)**.** *Given a finite set of abstract-predicates $\Psi = \{\psi_1, ..., \psi_n\}$, and a finite set of configurations of each location of XTG $Conf(l_j)$ where $j = 1, ..., m$ then the abstraction function α is defined by*

$$\alpha(\langle l, \rho\rangle)(j) := \langle l_j, \psi_i\rho_j\rangle$$

where $i = 1, ..., n$ and the concretization function γ is defined by

$$\gamma(\langle l, b\rangle) := \{\langle l, \rho\rangle \in Conf(\mathcal{X}) \mid I(l) \wedge \bigwedge_{i=1}^{n} \psi_i\rho_j \equiv b(i)\}$$

Now, we can see that the abstraction/concretization pair (α, γ) forms a Galois connection.

Example Consider again the small XTG system from Figure 4.1.(a). The system can be described with three sets of configurations for each three locations. Each location $l\_$ has three valuations denoted that a_{l1}, a_{l2}, and a_{l3}, where $l = 0, 1, 2$ in Figure 5.2.(a). We choose four abstraction-predicates, $\Psi = \{\psi_1, \psi_2, \psi_3, \psi_4\}$ in Figure 5.2.(b) in order to compute the initial abstract representation, which also can be described as an initial over-approximating abstract representation by the Backward process of IAR, which will be explained more in section 5.2.

Then the abstract node could be written as a bit-vector of length four with the four bits representing the true values of $\psi_1, ..., \psi_4$ respectively in a sense that ψ_i can be true if the element of ρ_l, where $l = 0, 1, 2$, holds for the i-th element of Ψ. See Figure 5.2.(c)-(1).

For example, considering ρ_0 with its elements a_{01}, a_{02}, a_{03} such that $a_{01} \models \psi_1$, $a_{02} \models \psi_2$, and $a_{03} \models \psi_3$, the XTG system can be represented in the abstraction sense with n_0 for its initial location $\langle l_0, \rho_0\rangle$. The n_0 is given by the bit-vector [1110] such that $\rho_0 \Vdash \psi_i$, $i = 1, 2, 3$ but there is no *true* valuation for ψ_4, i.e. $\rho_0 \nVdash \psi_4$. The second node n_1 has the same bit-vector like n_0.

Whereas, the n_3 is given by the bit-vector [0111] such that $\rho_2 \Vdash \psi_i$, $i = 2, 3, 4$ since there might exist *false* valuation for ψ_1. For example, when n_3 has a predicate label $c = 5$ then it has *true* valuation related to ρ_2, but the other case such as $c > 5$ then n_3 has *false* valuation. Thus we don't know whether or not ψ_1 is *true*. In fact we denote this undecided valuation as $\perp$. (see Figure 5.2.(b))

Obviously, we now can see that n_0 and n_1 have same bit-vector. Thus those two nodes can be abstracted into one n_{01}, which is depicted in Figure 5.2.(c)-(2).

In our PDT form, actually each node is labelled with a set of abstraction-predicates satisfies the condition $\rho \Vdash \psi$ in the node. Then it is easy to see that each node n_{01}, n_2 in Figure 5.2-(d) is corresponding to each node n_0, n_2 labelled by $0 \leq c \leq 5, 0 \leq x \leq 1, go = 0, \neg(c \geq 5)$ and $c \geq 5, 0 \leq x \leq 1, go = 0$ respectively in Figure 5.2.(d).

We have described the relation between abstraction and concrete model by Galois connection, and also showed how to compute the abstraction from the given concrete model with a small example. We now adapt these ideas to deal with the construction of complete abstract representation that will be explained in next section.

5.2 Iterative Abstract Refinement

In order to deal with the restriction of model checking based verification, building abstract models, which allows property-preserving transformations making models amenable to formal verification is the main principle of this thesis.

A promising approach to construct abstractions (automatically) is to map the concrete states of a system to abstract states according to their evaluation under a finite set of predicates. Many efforts have been made to construct predicate abstractions of systems [1, 9, 19]. Where do the predicates for predicate abstraction come from? A popular scheme for generating predicates is to guess a set of initial predicates, and use (spurious) counterexamples from a model checker to generate more predicates as necessary [28, 15, 11, 19]. Such schemes go by the name of counterexample guided abstraction refinement (CEGAR).

Predicate abstraction [31, 60] determines a finite abstraction, where each state of the abstract state space is a truth assignment to the abstraction predicates. The abstraction is conservative in the sense that a propositional formula holds for the concrete systems if it holds for the predicate-abstracted system. Since the reverse statement does not hold in general, predicate abstraction has so far mainly been used to only prove safety but not liveness properties.

Unlike previous works for the automatic abstraction of infinite state systems using the CEGAR approach, we propose a new and practical (semi-automatic) verification methodology in order to construct a complete abstraction enables us to prove safety and liveness properties of the concrete system with less complexity. To make our proposal concrete, we present a tool supported verification with combination of predicate abstraction, deduction and model checking techniques

$$\langle l_0, \rho_0 \rangle \equiv Conf(l_0) = \{a_{01}, a_{02}, a_{03}\}$$

$a_{01} \equiv c \leq 3$ $\quad a_{02} \equiv x = 0$ $\quad a_{03} \equiv go = 0$

$$\langle l_1, \rho_1 \rangle \equiv Conf(l_1) = \{a_{11}, a_{12}, a_{13}\}$$

$a_{11} \equiv 3 \leq c < 5$ $\quad a_{12} \equiv x = 1$ $\quad a_{13} \equiv go = 0$

$$\langle l_2, \rho_2 \rangle \equiv Conf(l_2) = \{a_{21}, a_{22}, a_{23}\}$$

$a_{21} \equiv c \geq 5$ $\quad a_{22} \equiv x = 1$ $\quad a_{23} \equiv go = 0$

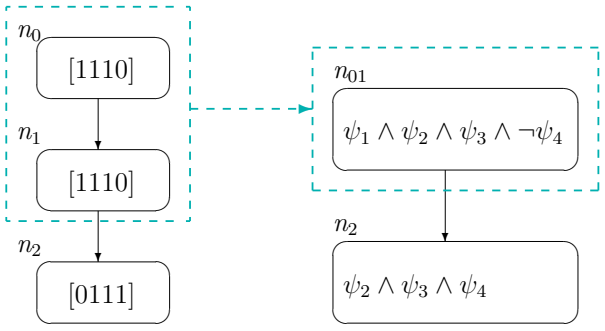
(a) Concrete system modification

$\Psi = \{\psi_1, \psi_2, \psi_3, \psi_4\}$ $\quad \psi\rho_0 = \{t, t, t, f\}$

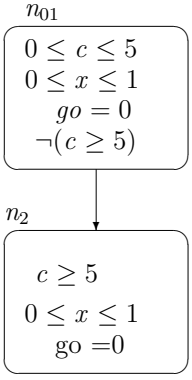
$\psi_1 \equiv 0 \leq c \leq 5$ $\quad \psi_3 \equiv go = 0$ $\quad \psi\rho_1 = \{t, t, t, f\}$

$\psi_2 \equiv 0 \leq x \leq 1$ $\quad \psi_4 \equiv c \geq 5$ $\quad \psi\rho_2 = \{\bot, t, t, t\}$

(b) Abstraction predicates and juxtapositions



(c) Abstract mapping-(1) (c) Abstract mapping-(2)



(d) Abstract representation

Figure 5.2: *abstract-representation*

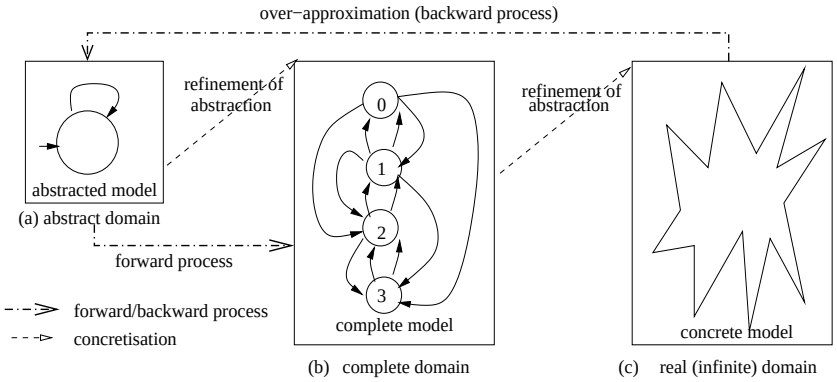


Figure 5.3: *Abstract, complete, and concrete models*

to verify a rich class of safety and liveness properties of timed systems. Actually it aims to compute a finite abstraction that guarantees its conformance w.r.t the concrete system by successive refinement. We call this methodology *Iterative Abstract Refinement* (IAR).

This methodology is based on a simple, efficient, and precise form of abstraction generation that preserves properties and guarantees the correctness of abstraction; The idea is to perform an initial over-approximation (often called skeletal abstraction in this thesis) of the model and then verify two propositions: (1)-the conformance between an abstract representation and its concrete model, and (2)-the evaluation required property on the result of proposition-(1). If the abstract representation conforms to its concrete model and the properties of interest can be successfully verified (or a positive result) over the abstract model, they also hold of the concrete system as well then we can say that such an abstract model is the complete abstract representation.

Before we talk about IAR, we illustrate three different models of a system. Figure 5.3 shows models over a skeletal abstract, a complete, and a concrete domain of a system.

The abstract model shown in Figure 5.3.(a) is obtained by over approximation; it can therefore not guarantee that the abstract model-(a) preserves corresponding runs of a concrete model-(c) in itself. The model of Figure 5.3.(b) obtained from an abstract model-(a) by iterative refinements until it conforms to the concrete model-(c) and the required properties can be successfully evaluated. We expect to obtain such a complete model-(b) using IAR.

Figure 5.4 (related Figure 5.3) shows the overall framework of IAR. The input to IAR consists of three parameters: the XTG specification of the concrete system, properties (desirable behaviors) to be verified, and a finite set of abstraction-predicates.

IAR has two (Forward/Backward) processes: first over-approximation (skele-

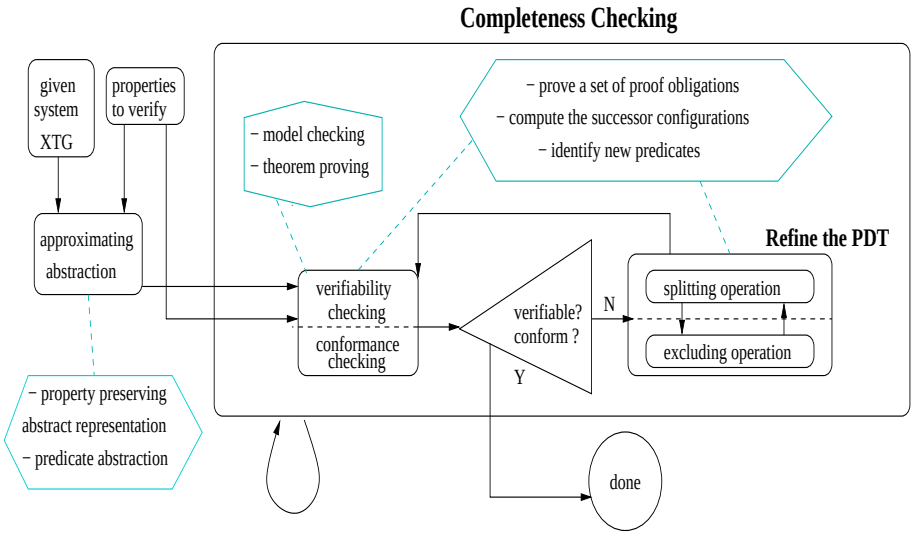


Figure 5.4: *Overview of IAR*

tal abstract representation) can be obtained manually from XTG by applying Backward process. Starting from the first skeletal abstraction, IAR iterates Forward process until the abstract representation becomes complete:

Backward process: direction (c) to (a) in Figure 5.3 A trivial and incomplete approximation is extracted from XTG by predicate abstraction. The result abstract representation is depicted with a PDT, which is an abstract representation form of XTG from Chapter 4 and how this extraction performed was shown in Figure 4.1.(a) in section 4.7 with a small example. However, this skeletal PDT by Backward process does not guarantee the completeness condition since each path allowed by XTG cannot be traced through it. In case this Backward process alone is impractical to generate complete abstract representation. Thus, another process, called Forward process, is required to refine such incomplete abstract representations.

Forward process: direction (a) to (b) in Figure 5.3 IAR picks a PDT and – if the PDT appears to be complete by showing that properties are verifiable and preserved over the PDT and the PDT conforms to XTG – infers correct abstract representation and successful verification, or – if the PDT is not complete – enforces completeness (refines the PDT) by two operations: a *split* operation and an *excluding* operation.

Two operations are applied while IAR pursues the completeness of abstract representation (PDT) from the Backward process by checking – (1) *verifiability* of properties on the PDT – (2) *conformance* between the PDT and XTG.

- If IAR fails to prove verifiability and conformance then IAR splits the PDT w.r.t. abstraction-predicates in order to enrich the PDT as adding details in the PDT.
- During the operation, a set of proof obligations (a number of verification conditions expressed in first-order logic) are proved in order to exclude extra duplicated transitions among PDT nodes caused by splitting. Splitting and excluding are continued until the PDT becomes complete.

To give a taste of how such two operations work, we remember again about the small abstract representation example in Figure 4.1.(a), which shows that the splitting and excluding operations would occur, when dealing with the correctness of abstract representation in our conformance checking based approach. During the refinement of (a) in Figure 4.1 for the reason that Figure 4.1.(a) could not be verifiable for the given property (ref. an example in Chapter 4), the initial node n_0 was split into two nodes n_{01} and n_1. Note that according to our definition of PDT, locations are not represented by predicates and they are special. In fact we only like to show how our two operations work with a PDT, therefore in this Figure 5.5, we do not consider the special labels for locations.

Because of the splitting, three possible transitions are added in Figure 5.5 and the result is shown in Figure 5.5.(b). The three possible transitions can be formulated by regarding the second condition of Theorem 4.5 and needed to check the validity for each of them:

$$n_{01} \wedge \mathit{Next} \stackrel{?}{\Rightarrow} n'_1 \tag{5.1}$$

$$n_1 \wedge \mathit{Next} \stackrel{?}{\Rightarrow} n'_{01} \tag{5.2}$$

$$n_{01} \wedge \mathit{Next} \stackrel{?}{\Rightarrow} n'_2 \tag{5.3}$$

Where, $n\_$ is a set of labels for each node and $n'\_$ is a set of labels for its post node, also *Next* is a set of conjunction of invariants of all active locations before and after the transition of $\mathcal{X}$ corresponding to $n\_$.

For the transition-(5.1), we obtain proof obligation

$$\begin{array}{ll} & x = 0 \wedge go = 0 \wedge 0 \leq c \leq 3 \wedge c = 3 \wedge x' = x + 1 \wedge go' = go \wedge c' = c \\ \Rightarrow & x' = 1 \wedge go' = 0 \wedge 3 \leq c' \leq 5 \end{array}$$

hold. In the same way, for the transition-(5.2), we obtain proof obligation

$$\begin{array}{ll} & x = 1 \wedge go = 0 \wedge 3 \leq c \leq 5 \wedge c' = 0 \wedge x' = 0 \wedge go' = go \\ \Rightarrow & x' = 0 \wedge go' = 0 \wedge 0 \leq c' \leq 3 \end{array}$$

hold as well. On the other hand, for the transition-(5.3) we obtain proof obligation

$$\begin{array}{ll} & x = 0 \wedge go = 0 \wedge 0 \leq c \leq 3 \wedge x' = x + 1 \wedge go' = go \wedge 3 \leq c' \leq 5 \\ \not\Rightarrow & x' = 1 \wedge c' \geq 5 \wedge go' = 0 \end{array}$$

doesn't hold. Thus, we know the transition-(5.3) is an extra duplicated transition should be excluded. In the Figure 5.5.(b), this extra transition is illustrated with a "False" mark. Finally, after leaving out of the "False" transition, we have the same diagram in Figure 4.1.(b) in Chapter 4. Then now we do model checking over this diagram for establishing a property we like to verify.

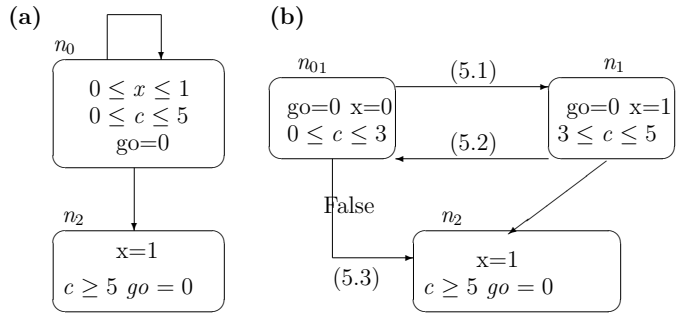


Figure 5.5: *An example of splitting operation*

IAR guarantees its termination when it results that a current Δ has corresponding paths of the $\mathcal{X}$'s so that properties verified by the Δ are preserved on $\mathcal{X}$; more specifically, from the first approximation, IAR refines the incomplete abstract representation by adding possible new nodes and edges corresponding to $\mathcal{X}$ more and more in detail. This is continued until –(1) properties presented in the PDT are successfully verified –(2) the current Δ can find no more new nodes and edges to be added.

In a nutshell, IAR constructs the abstract representation from concrete model and justifies its completeness by combination of verifiability and conformance checking. In the rest of this chapter, we investigate how the PDT resulted from IAR leads to the verification framework.

Again, in this thesis counterexample-guided abstraction refinement (CEGAR) part is not really required since during the refinement with conformance checking, we use safety property (desired behavior) as one of predicates that can construct abstract model which already guarantees that PDT captures all runs of XTG including the desire behavior. However, we believe that major improvements could be made here by applying CEGAR resulting in a much higher degree of refinement technique.

5.3 Verification for Safety properties

In infinite state systems, safety properties can be proved by induction [48]. First an inductive invariant has to be proved on the system. This means that the invariant holds in the initial state of the system and every possible transition

preserves it, that is if invariant holds in some state then it continues to hold in every successor state as well. Now if the inductive invariant implies the desired property then the proof is complete.

Consider the finite state system (PDT) in which there is a Boolean state variable for each of the atomic predicates present in the inductive invariant. Given a sate in the original system, the corresponding state in the finite state system can be computed by evaluating each of the predicates and then constructing a bit vector out of the results. A pair of states in the finite state system are in the abstract transition relation if actual states corresponding to them are in the transition relation of the original system.

5.3.1 PDT for Fischer's protocol and its safety

In this section, we take Fischer's real-time mutual-exclusion protocol [8, 42] in order to illustrate our IAR approach by verifying safety property. We take the simplified version with only two processes in the protocol. Figure 5.6. shows the structure of process i, where $i = 1, 2$, and k is a shared variable accessed by both processes, whereas c_i is a local clock of the process. The timing constraints make sure that all processes at location 2 (l_{i2}) wait until all processes at location 1 (l_{i1}) make the transition to location 2 (l_{i2}).

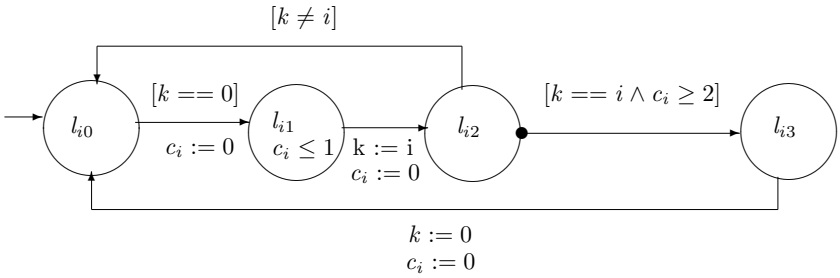
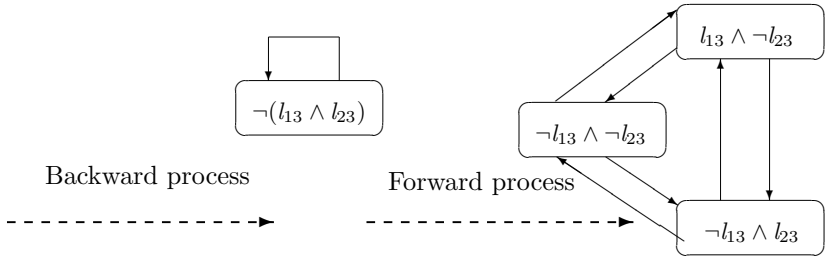


Figure 5.6: *XTG specification for Fischer's protocol*

Intuitively, the protocol behaves as follows: in the first phase each process tries to register its process identification in the shared variable k. In the second phase each process tests whether its identity is still registered in k after a predefined lapse of time and then enters the critical section. The purpose of the protocol is to ensure its safety such that there is never more than one process in the critical section, expressed by the LTL formula

$$\Box\neg(\mathbf{at}\, l_{13} \wedge \mathbf{at}\, l_{23}) \tag{5.4}$$

As the whole procedure of IAR, we first have to specify the protocol in $\mathcal{X}$ shown in Figure 5.6. From this $\mathcal{X}$ we extract the first Δ by Backward-process as concerning the LTL formula 5.4 which described the property that both processes may never



(a) The first PDT of XTG (b) The forward-refinement of (a)

Figure 5.7: *The backward and forward approach for Fischer's protocol*

be at the beginning of their critical section, location 3 (l_{i3}) and the first PDT Δ is shown in Figure 5.7.(a) that preserves the property.

The Forward process is applied to Figure 5.7.(a) in order to enrich Figure 5.7.(a) until it becomes complete. For building the complete Δ, we consider two aspects such that – if the desired property is decidable over Δ, and – if Δ correctly conforms to $\mathcal{X}$ (accuracy of representation). In fact we see the first aspect is already satisfied with a Δ in Figure 5.7.(a) extracted by Backward process as preserving the desire property.

Now we consider the second aspect, accuracy of representation. The first Forward process on Figure 5.7.(a) results in Figure 5.7.(b); Although the simple diagram shown that the desire bahaviour of system (high level property) is satisfied in Figure 5.7.(a), the degree of conformity of a measured or calculated quantity to its actual (true) system is very low.

Thus, we need to enrich this (a) to improve its accuracy to $\mathcal{X}$. The first step towards Forward refinement is implemented by the diagram shown in Figure 5.7.(b), which rules out direct hand-overs of access of the critical section from one process to the other. We no longer show the loops on the nodes because they are implicit in the definition of a run. It is easy to see that the diagram (b) is a structural refinement of (a) because every node label implies $\neg(l_{13} \wedge l_{23})$ and every transition is allowed by the loop on the only node of the initial diagram.

However, this (b) is still not detailed (precise) enough and neither does it satisfy any liveness property. We now see some events and invariants (predicates) would be added into this (b) in order to enrich (b) towards constructing a complete Δ, which is a correctly conformed Δ w.r.t $\mathcal{X}$, and the complete Δ enables to verify safety/liveness properties over itself. We infer new nodes would be added by tracing runs of $\mathcal{X}$. In order to infer those runs we use conformance conditions of Theorem 4.5.

For instance, we first consider an initial node of Δ that satisfies conditions of the set of initial locations of $\mathcal{X}$, i.e. the initial node should contain a set of

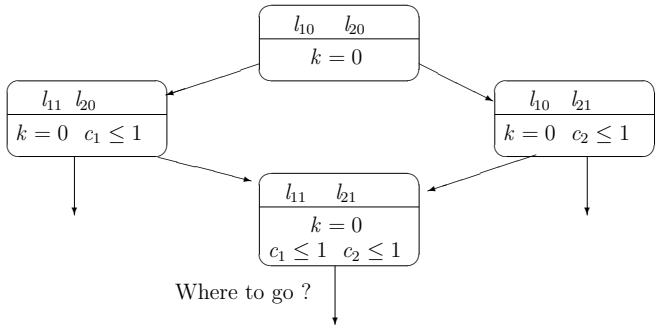


Figure 5.8: *The partially refined model of (b) (cf. Figure 5.7).*

cofigurations of initial control locations of $\mathcal{X}$. As knowing conditions of initial locations l_{10} and l_{20}, we see easily that the initial node contains $k = 0$ as its label and at the same time this initial node implies $\neg(l_{13} \wedge l_{23})$. To ensure this node is a correct one, we check its conformance with $\mathcal{X}$ by a condition 1 of Theorem 4.5:

$$k = 0 \wedge l_1 = 0 \wedge l_2 = 0 \Rightarrow k = 0$$

we see this formula obviously holds.

Thus, from this initial node we infer a set of post nodes by analyzing a set of possible successors of l_{10} and l_{20}: there are two possible transitions from l_{10}, l_{20} to either - (1) l_{11}, l_{20} or - (2) l_{10}, l_{21} that have to be matched with edges leaving from the current initial node. This condition can be formulated regarding Theorem 4.5 as following:

1. $k = 0 \wedge c1' = 0 \wedge k' = k \wedge c1' \leq 1 \quad \models \quad N'_{post}$
2. $k = 0 \wedge c1' = 0 \wedge k' = k \wedge c2' \leq 1 \quad \models \quad N'_{post}$

where, N'_{post} is a set of predicate labels of post node of current node. Given formula can be validated by adding new predicates to its right hand side to make the whole formula true. Thus the problem is now focusing on finding appropriate predicates that make given formula true. We can easily find N'_{post} for (1) and (2) by seeing the corresponding formula become

1. $k = 0 \wedge c1' = 0 \wedge k' = k \wedge c1' \leq 1 \quad \Rightarrow \quad k' = 0 \wedge c1' \leq 1$
2. $k = 0 \wedge c1' = 0 \wedge k' = k \wedge c2' \leq 1 \quad \Rightarrow \quad k' = 0 \wedge c2' \leq 1$

and the new predicates on the right hand side $k' = 0 \wedge c1' \leq 1$, and $k' = 0 \wedge c2' \leq 1$ is to be added into post nodes of initial one. In Figure 5.8, we see four nodes and their predicates with edges found by being inferred and analysed w.r.t corresponding transitions and locations of $\mathcal{X}$. In fact this model in Figure 5.8

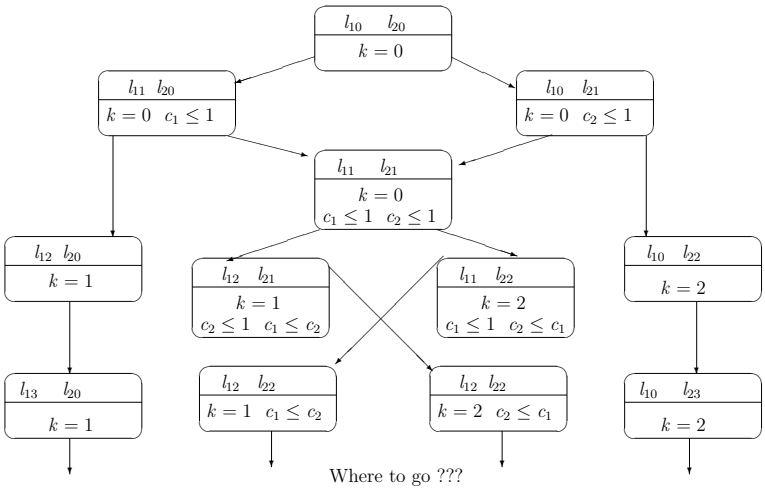


Figure 5.9: *The partially refined model of (b) (cf. Figure 5.7).*

is not a PDT form but a partially refined model of (b). This one is a part of consequence on the way of building a complete PDT.

In order to have a fully refined Δ, we keep on applying the same procedure to explore post nodes of current one by adding more events and invariants that indicate which process has issued a request, and which process has become a winner as diagnosing the corresponding runs of $\mathcal{X}$. Such a mechanism is marked as "where to go" in Figure 5.8. We see more nodes and edges are explored and added in Figure 5.9, and at the same time those nodes implies $\neg(l_{13} \wedge l_{23})$ as well.

Finally, we reach a Δ assumed that the desirable behaviour of $\mathcal{X}$ consumed in the Δ is shown as Δ_1 in Figure 5.10.

Again we have a question: is it conforms to $\mathcal{X}$? The answer is surely "yes" since during new nodes (post nodes) exploration, we applied Theorem 4.5 to indicate necessary predicates of post nodes and their edges by regarding the fact that every behaviour of the Fischer protocol specified in $\mathcal{X}$ is a trace through Δ_1. For sure, we can pick up any possible node and its edge from Δ_1 to show that Δ_1 conforms to $\mathcal{X}$ by discharging the conditions of Theorem 4.5.

As an example, we consider the possible transitions of process 1 from the node marked (*) in Δ_1 with corresponding control locations l_{12} and l_{23}. There are two possible transitions: the first leads from l_{12} to l_{10} and is represented by the edge to the right neighbor of the marked node in Figure 5.10. Indeed, the corresponding proof obligation

$$k = 2 \wedge k \neq 1 \wedge k' = k \wedge c_1' = c_1 \wedge c_2' = c_2 \Rightarrow k' = 2$$

is obviously correct. The other possible transition of process 1 in $\mathcal{X}$ corresponds to a move to the critical section (location l_{13}). Because no matching node is

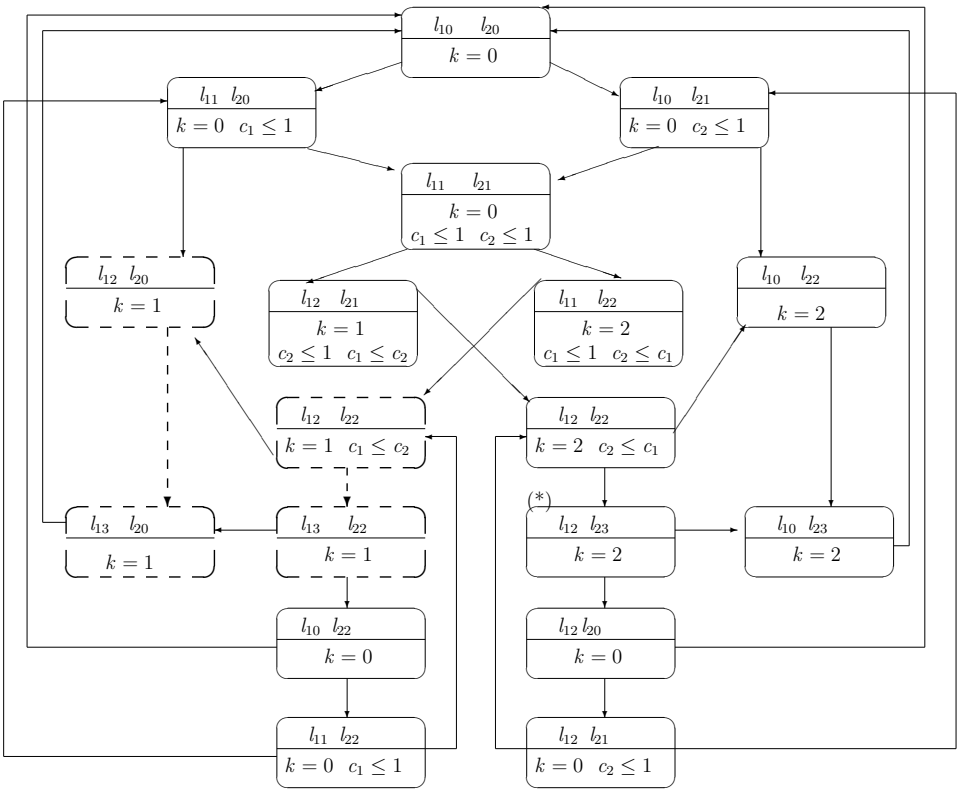


Figure 5.10: Δ_1 *for Fischer's protocol (cf. Figure 5.6).*

reachable in Δ_1, the proof obligation becomes

$$k = 2 \wedge k = 1 \wedge c_1 \geq 2 \wedge k' = k \wedge c_1' = c_1 \wedge c_2' = c_2 \;\Rightarrow\; \textbf{false}$$

which holds because the left-hand side is contradictory. Effectively, we demonstrate that process 1 cannot enter when process 2 is already inside its critical section. The remaining proof obligations are similar and they are proven by CVC-LITE. Note the complete proof is done in Appendix A. (Observe that Δ_1 of Figure 5.10 contains no time-passing edges other than the self-loops, which we do not show explicitly according to our convention)

The verification can be supported by two ways such that method-(1) is using the DIXIT toolkit by model checking SPIN or PMC model checker. For our example, DIXIT reports that the diagram satisfies mutual exclusion. The other one, method-(2) is that alternatively, we can use CVC-LITE again to infer the property of interest from each of the node labels. For instance, to check the satisfiability

of the formula 5.4, we query a following alternative formula

$$\neg l_{13} \vee \neg l_{23} \tag{5.5}$$

to every node of Δ_1 using CVC-LITE. Because every node of Δ_1 satisfies a formula 5.5, we conclude that Fischer's protocol ensures mutual exclusion.

5.4 Verification of Liveness property

We have constructed a Δ which is conformed to $\mathcal{X}$ and verified safety property over the Δ so far. In this section, we consider how to verify liveness properties using the Δ.

5.4.1 Auxiliary conditions of PDTs

The correct formulation of liveness requires us to focus on particular process and to stipulate that a request of the process is always eventually acknowledged.

Verifying such a liveness property requires additional considerations. Indeed, XTG specifications ensure liveness by using clock constraints such as urgent conditions (marked by "asap") associated states and transitions. Let us recall that each node of a PDT has a τ-loop implicitly that allows an infinite stuttering in some nodes which should not. Thus, the PDT describes previously does not allow immediately to check the liveness property.

To overcome this, we add an auxiliary condition to the Δ; each node in Δ labels extra clock-constraint predicates. Assume that we look at a single node has any trace has to leave from the node. In case we enable this trace as forcing not to cause infinite stuttering in a way that we set an upper-bound clock as a clock constraint in the node. The node has such a constraint condition called *timed-bounded node*

Definition 5.3 (Time-bounded node)**.** *Let $\Delta = \langle N, N_0, R_\mu, R_\tau \rangle$ be a PDT over L and $\mathcal{P}$. A node $n \in N$ is a time-bounded node if its label has a predicate of the form $c \leq e$, where $c \in V_c$ is a clock and $e \in Expr$ is an expression. The constraint $c \leq e$ is indicated by $clk(N)$ and the expression e is denoted by $\uplus(N)$*

The mentioned condition in Definition 5.3 can be viewed as requirements equipped with some time-constraints as labels in nodes that are used to represent "asap" conditions of $\mathcal{X}$; it introduces a concept of time-progess in the analysis of the PDT; it is related to the invariants of the states of the XTG and the urgent conditions.

5.4.2 PDT for Fischer's protocol and its liveness property

In this section, we exemplify verification of liveness property mentioned in previous section with a same example, Fischer protocol. Recalling some specific

behaviour of this Fischer protocol such that all processes at l_2 wait until all processes at l_1 make the transition to l_2 in $\mathcal{X}$. To avoid an unexpected case that a process stays at l_2 forever, an appropriate liveness condition should be considered as well. Hence, the liveness property can become that any process at l_2 eventually enters at l_3.

However the Δ_1 we have implemented so far is still weak to handle the liveness property: it does not represent the urgent condition, which the corresponding $\mathcal{X}$ has. The urgent condition depicted with a small black dot on the transition from l_2 to l_3 in Figure 5.6.

Continuing our development towards a goal that we verify the liveness property over Δ_1, we add some clock-constraints into each node in Δ_1; i.e we reinforce Δ_1 by giving auxiliary conditions (see Definition 5.3). After this modification, these nodes become time-bounded nodes.

Corresponding to the urgent transitions coming from at l_{i2} of $\mathcal{X}$ (see Figure 5.6), we add the constraints $c \leq 2$ to the nodes of Δ_1 corresponding to these control locations. Formally (see Figure 5.11), we put

- $\uplus(n) = 2$ for $n \in \{n_{L1}, n_{L3}, n_{R1}, n_{R3}\}$,
- $clk(n) = c_1 \leq \uplus(n)$ for $n \in \{n_{L1}, n_{L3}\}$,
- $clk(n) = c_2 \leq \uplus(n)$ for $n \in \{n_{R1}, n_{R3}\}$.

hese additions of predicates, which correspond to the location invariant of the XTG, are justified by the proof obligations of the conformance theorem, Theorem 4.5, where the location invariants appear on the left-hand side of the implication.

Based on the information about upper-bound clocks and clock-constraints, we can label those clock-constraints to the suitable nodes as time-invariants. For instance, let's look into some transitions of process 1 from the node has a dashed frame and an edge in Δ_1 with corresponding location l_{12} and l_{20}. In fact the transition is taken from l_{12} to l_{13} should have a matching edge, which enables a run from/to the corresponding nodes. Therefore the nodes with dashed frames and edges should have clock-constraints, and the nodes become time-bounded nodes. The result is shown in Fig 5.11 as PDT Δ_2.

The result Δ_2 in Figure 5.11 shows a diagram with the same structure as the Δ_1 in Figure 5.10, but with an additional labelling of the nodes. As we justified before, it is easy to show that the Δ_2 also conforms to $\mathcal{X}$; it imposes a constraint to avoid infinite stuttering in nodes which is excluded. A formal proof of conformance using CVC-Lite is shown in the Appendix A.

The verification of liveness property using Δ_2 is possible by using Dixit toolkit. This property is expressed in LTL as follows:

$$\Box\left(\left(\bigvee_{i=1}^{2} l_{i2}\right) \Rightarrow \Diamond\left(\bigvee_{i=1}^{2} l_{i3}\right)\right) \tag{5.6}$$

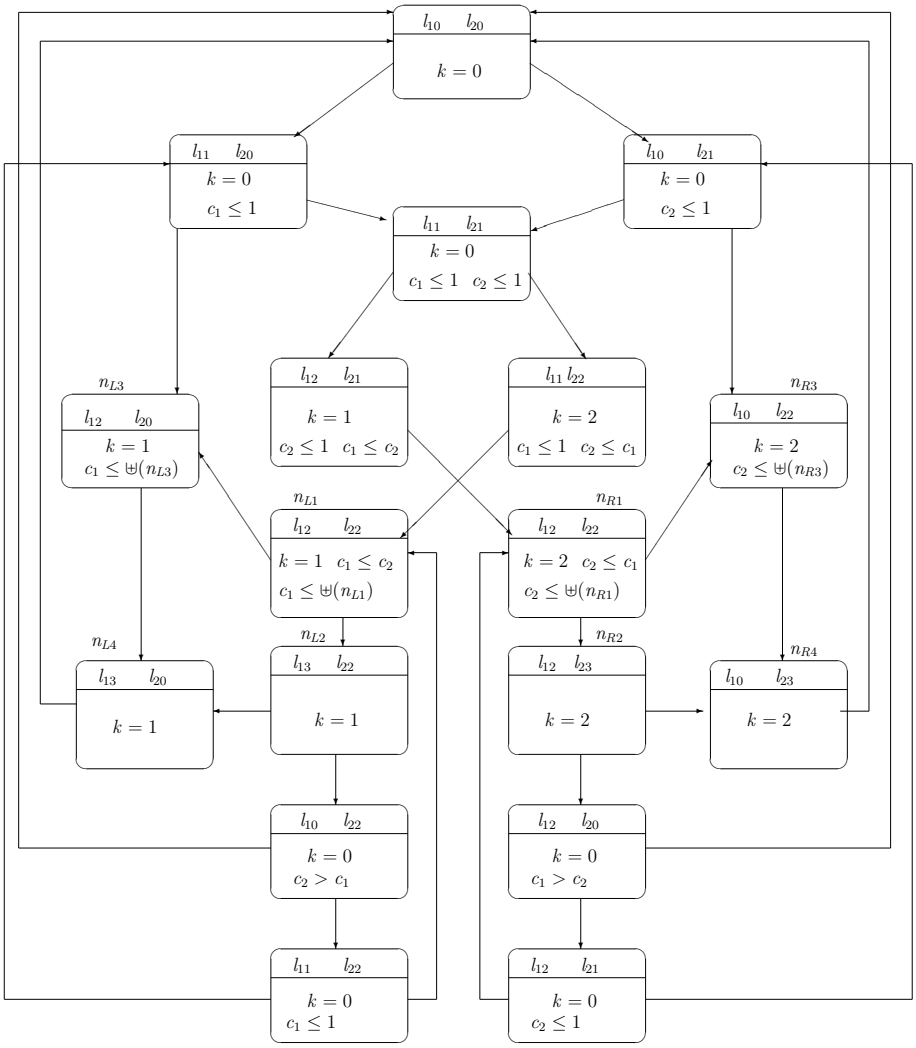


Figure 5.11: Δ_2 *for Fischer's protocol (cf. Fig. 5.6).*

It will be observed that the tool CVC-Lite is not useful for checking this formula because it does not have pure "expressions" or "commands" for liveness formula such as $\Diamond$ (in words "eventually" or "in the future"). In case we do the verification by help of DIXIT toolkit or PMC model checker; Concerning DIXIT, we need to translate the time-boundedness condition into appropriate fairness hypothesis for model checking: first, the LTL property 5.6 is inpterpreted to the formual which is checkable by DIXIT and we add the time-boundeness condition to the formula as time contraints $(c_1 \leq 2, c_2 \leq 2)$ as follows:

$$\Box\Bigg(\Big((l_1 = 2 \wedge c_1 \leq 2) \vee (l_2 = 2 \wedge c_2 \leq 2)\Big) \Rightarrow \Diamond\Big(l_1 = 3 \vee l_2 = 3\Big)\Bigg) \tag{5.7}$$

We require strong fairness of the edges for $\{(n_{L1}, n_{L2}),\ (n_{L3}, n_{L4}),\ (n_{R1}, n_{R2}),\ (n_{L1}, n_{L2}),\ (n_{R3}, n_{R4})\}$, and weak fairness of the edges for $\{(n_{L1}, n_{L3}),\ (n_{R1}, n_{R3})\}$ in order to ensure that stuttering behaviours are resolved in a fair way, and that a process is at l_2 with a timed constraint that will enter the l_3. Model checking establishes that the diagram satisfies the liveness properties.

In this framework discussed here with Fischer protocol example has not considered about CEGAR approach since we introduced a formal intermediate layer proving the correctness of abstraction by conformance checking. This layer allows property-preserving abstract representation making the model amenable to model checking; we use the property interest (desirable behaviors) as abstraction predicates in the sense that the constructed abstract representation (result Δ) captures the run of desirable behaviour of $\mathcal{X}$. Thus, we can know that applying model checking on the result Δ doesn't give false-negative since Δ is constructed by making the property we like to verify *true*.

5.5 Conclusion

In this chapter we have introduced IAR, which is the method of combining predicate abstraction, theorem proving and model checking techniques based on abstract refinement that constructs a correct abstract representation from a concrete system and justifies the completeness of it.

We have also shown that some desired properties, not only for safety but also liveness property, can be verified over the given abstract representation (PDT) constructed by IAR. We defined auxiliary conditions that strengthen the PDT to be able to prove liveness property.

Fischer mutual exclusion protocol example to which the IAR and verification methods have been applied are presented.

In fact, the work as described in this chapter is restricted to deal with real time systems with finite control only. The predicate abstraction of timed systems, however, can readily be extended to also apply to richer models such as parameterized timed automata and even to timed automata with other infinite type such

as counters or stacks. In next chapter we will show how such extensions work as well.

Chapter 6

Predicate Diagrams for Parameterized Systems

Parameterized systems have become a very important subject of research in computer aided verification regarding the issue of decidability: Can we verify a system parametrically, for arbitrary N, instead of fixing a concrete value?

A typical parameterized system consists of an *arbitrary* number of identical processes interacting via synchronous or asynchronous communication. A typical example is a mutual exclusion protocol for an arbitrary number of processes competing for a common resource. Many distributed protocols that consist of parallel compositions of many identical processes, and control communication and synchronization of network systems are also examples as well.

A rising challenge is to design a method for the uniform verification of such systems; conventional model checking techniques can be used to verify instances of parameterized systems for fixed parameter values, for example for a fixed number of participant processes. In order to prove the correctness for any parameter value, we need to construct a single syntactic object that represents (an abstraction for) the entire family of systems, i.e. prove by a *single* proof that the system is correct for any value of the parameter.

The capability to build a kind of "uniform" verification of parameterized systems is one of the impacting advantages of the deductive method for temporal verification over the model checking technique; model checking can be used to verify the desired properties of the systems for specific values of N such as $N = 3, 4, 5$.

Usually, the model checker's memory is not capable for values of N larger than 100 [46]. Furthermore, in the specific verification case we take with well-known scalable benchmark problem, namely the Fischer's mutual exclusion, given in Figure 5.6, shows how strongly model checkers are affected by the values N during the system verification.

The example is chosen again for various reasons. The example is well studied within the community of researchers in the context of verification of real-time systems. Secondly, it is simple enough to illustrate the power of XTG $\mathcal{X}$ and PDT Δ constructions. But more importantly, it shows how the verification problem becomes complex towards the number of processes N in the system.

The size of the example is simply scaled up by adding more processes in the protocol, with the result that the control space becomes more complex and the number of clocks increases. These two cause the phenomenon from model-checking context known as state explosion that obviously seen by following experiments.

The verification experiment was performed on a 1GHz Pentium III with 512 Mb internal memory on Linux by using the real-time model checkers PMC [12, 56] and Uppaal [43]: Uppaal is currently the most mature model checking tool for verifying real time systems. Its development started in 1995 and since then many people have worked on it, resulting in a much used and relatively mature verification tool. Another model checking tool called PMC (Prototype Model Checker) aimed at real-time concurrent systems is able to efficiently deal with especially XTG specifications $\mathcal{X}$ that use only simple constraints and updates.

	PMC		Uppaal		Uppaal [-A]	
	time	mem	time	mem	time	mem
Fischer-2	0.04	1.3	0.02	2.4	0.04	2.0
Fischer-3	0.04	1.3	0.03	3.3	0.04	2.0
Fischer-4	0.05	1.3	0.05	3.3	0.06	2.0
Fischer-5	0.17	2.8	0.68	4.7	0.13	3.3
Fischer-6	1.4	6.8	143	26	0.41	4.2
Fischer-7	17	32	-	-	1.7	5.8
Fischer-8	430	243	-	-	7.2	16
Fischer-9	-	-	-	-	30	41
Fischer-10	-	-	-	-	125	145

Table 6.1: *Verification Performance of Fischer-protocol by several model checkers*

The result is shown in Table 6.1 and it indicates that verification is infeasible for more than 10-processes in terms of required resources – either the system ran out of memory or verification was terminated after 15 minutes. For Uppaal we performed the test for two cases, without or with approximation option. The latter is enabled by -A option.

The figures first of all show that the relative performance of tools is strongly related to the type of verification problem: PMC fails already at after Fischer-8. When Uppaal uses its approximation option, which seems to lead to a better performance, however it cannot go further than Fischer-10.

Unlike those handicaps model checkers have, our IAR method we have applied on PDT Δ is to be able to establish the validity of the property for any value of N processes based on the integration of deductive and model checking techniques.

The using of IAR as applying PDT in the verification of parameterized systems will be shown in following sections. To show that, we consider two classes of properties: properties of the entire system, i.e. concerning all processes, and "per-process" properties that should hold of every single process in the system. The latter properties are sometimes referred to as *universal* properties. Given a parameterized system that consists of N processes, universal properties are expressed as formulas of the form $\forall i \in 1, \ldots, N : P(i)$.

We first mention about related work. Then we define a parameterized XTG and PDT, prove conformance conditions. We explain the use of PDT for the verification of Fischer's mutual exclusion protocol related to the entire system. We also verify the property of a single process in the Fischer's protocol using PDT.

6.1 Related work

Theorem prover helps to guide for verification of parameterized systems or the verification for parameterized system often done by hand [50, 46, 33]. Several methods based on manual constructions of a process invariant are proposed [20, 58]. However it is not in general possible to obtain a finite-state process invariant since the general problem is undecidable [7].

In our study we have restricted to a class of parameterized systems that are interleaving and consisting of arbitrary number of components (N-processes) of timed systems. The parameterized systems are represented as PDT. The verification is done deductively and algorithmically by means of the PDT. This PDT can be viewed as the abstract representation of parameterized systems, i.e. we represent a family of processes in a single diagram.

The same idea but not for real-time is the work from Baukus et al. [13]; They have considered techniques for the verification of *universal* properties of parameterized systems based on the transformation of an infinite family of systems into a single transition system expressed as a formula in WS1S and applying abstraction techniques on this system.

6.2 Parameterized XTGs

We now investigate how to specify these systems in our modelling language. In the whole discussion, N denotes a finite and non-empty set of processes running in the system being considered.

As mentioned before, in the context of parameterized systems, the systems consist of many identical processes, precisely, they consist of the same transitions and the same liveness properties. As an example in Figure 6.1 a typical parameterized system, consists of a collection of an abtrary but finite number of finite-state components via interleaving parallel excution of each component

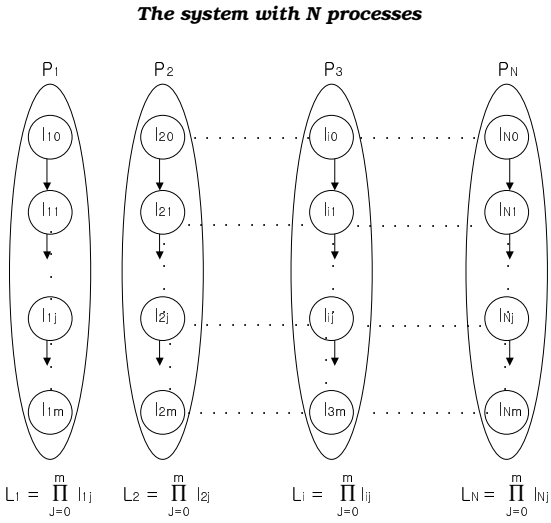


Figure 6.1: *System with N processes: N is a parameter of the protocol*

(process P_i), is depicted: the system is composed by N processes by running its algorithm (protocol) and each process consists of j number of control locations. N is a parameter of the protocol and L_i denotes a set of control locations of i-th process.

We start by defining a parameterized version of XTG. The structure communications becomes important if the system of XTG contains functions for the exchange of values and synchronization. (see Definitions 3.10 and 3.9).

Definition 6.1 (Parameterized XTG)**.** *A parameterized XTG, consists of an arbitrary but finite number of identical processes with m locations per each process and interacting via synchronous (or asynchronous) communication function Ch such that* $\mathcal{X} = \langle Init, \langle P_1, \ldots, P_n \rangle, Ch \rangle$ *with* $P_i = \langle Init_i, L_i, l_{i0}, I_i, E_i, U_i \rangle$*, is also an XTG. The semantics of* $\mathcal{X}$ *is given by the definition 3.11*

6.3 Parameterized PDTs

In this section, we will study the use of PDTs in the verification of parameterized systems. Recalling the definition of PDTs, Definition 4.2, PDTs are defined relatively to two sets, namely a set of nodes N and a set of the relations R. In the context of parameterized systems we can say that N now contains the (name of) parameterized predicates. Also, reminding the definition of standard predicate diagrams, Definition 4.1, a relation may have an associated fairness condition and

an ordering annotation $(\eta, \prec)$.

In the definition 4.2 we introduced PDTs with a set of control location L and a set of abstraction predicates $\mathcal{P}$. In the context of parameterized systems we will extend $\mathcal{P}$ so that it represents parameterized predicates. Given a parameterized XTG, the set of $\mathcal{P}$ contains not only usual predicates but also *parameterized predicates* by the identifier of the processes, i.e. *quantified predicates* for some execution of any process we want to quantify over the process identification.

Definition 6.2 (quantified predicates)**.** *A quantified predicate is one of the two forms*

- $\forall i \in 1, \ldots, N : f(i)$ *or*
- $\exists i \in 1, \ldots, N : f(i)$

where, $f(i)$ is a formula (not temporal) without quantifier with a parameter i which represents the identifier of a process. The whole quantified predicates will be indicated by $\mathcal{P}_Q$.

The formal definition of parameterized PDT is given with respect to a set L that represents locations of the underlying parameterized XTG, as well as with respect to a set $\mathcal{P}_Q$ of quantified predicates of interest. We write $\overline{\mathcal{P}_Q}$ to denote the set containing the predicates in $\mathcal{P}_Q$ and their negations.

The following definition of parameterized PDT is an annotation related to the assumptions of fairness and ordering relations. For the last condition, we suppose a finite set $\mathcal{O}$ of binary relation symbols $\prec$ that are interpreted by well-founded orderings. We denote by $\preceq$ its reflexive closure, and by $\mathcal{O}^=$ the set of relations in $\mathcal{O}$ and their reflexive closure. These extensions are transposed literally from the original work on the predicate diagrams of non-timed systems [17, 18].

Definition 6.3 (Parameterized PDT and run)**.** *Assume given finite sets L and $\mathcal{P}_Q$, an* parameterized PDT *over L and $\mathcal{P}_Q$ is a tuple $\Delta_p = \langle N, N_0, R_\mu, R_\tau, o, \Theta \rangle$ as follows:*

- *$N \subseteq L \times 2^{\overline{\mathcal{P}_Q}}$ is a finite set of nodes.*
- *$N_0 \subseteq N$ is the set of initial nodes.*
- *$R_\mu, R_\tau \subseteq N \times N$ are two relations represent discrete and time-passing transitions of the parameterized XTG. The relation R_τ is supposed be reflexive.*
- *o is a relation labelling that associates a finite set of term η_i $\{(\eta_1, \prec_1), \ldots, (\eta_h, \prec_h)\}$, with paired relations $\prec_i \in \mathcal{O}^=$.*
- *$\Theta : R_\mu \rightarrow \{NF, WF, SF\}$ associates a fairness condition with discrete transitions; the possible values represent no fairness, weak fairness, and strong fairness.*

A run *of* Δ_p *is an infinite sequence*

$$\sigma \quad = \quad n_0 \xrightarrow{lab_0} n_1 \xrightarrow{lab_1} n_2 \ldots$$

where $n_0 \in N_0$, $lab_i \in \{\mu, \tau\}$ *and* $n_i \rightarrow_{lab_i} n_{i+1}$ *for all* $i \in \mathbb{N}$ *such that the following conditions hold:*

- *for any transition* $(n, n') \in R_\mu$ *such that* $\Theta(n, n') = WF$ *there are infinitely many* i *such that either the transition* $n \xrightarrow{\mu} n'$ *appears infinitely often in* σ, *or* $n_i \neq n$,

- *for any transition* $(n, n') \in R_\mu$ *such that* $\Theta(n, n') = SF$, *either the transition* $n \xrightarrow{\mu} n'$ *appears infinitely often in* σ *for infinitely many* i, *or there are only finite many* i *such that* $n_i = n$,

- *for any pair* $(\eta, \prec)$ *of an expression* η *and an irreflexive relation* $\prec$, *if there is an infinitely many number of transitions* $n_i \xrightarrow{\mu} n_{i+1}$ *in* σ *with* $(\eta, \prec) \in o(n_i, n_{i+1})$ *then there exists an infinitely many number of transitions* $n_j \xrightarrow{\mu} n_{j+1}$ *in* σ *with* $(\eta, \prec) \notin o(n_j, n_{j+1})$ *and* $(\eta, \preceq) \notin o(n_j, n_{j+1})$.

In addition to the Definition 6.3, if node n_i has an outgoing edge with an ordering annotation $(\eta, \prec)$, stuttering transitions are forbidden to increase the value of η. Fairness conditions are used to prevent infinite stuttering.

A parameterized PDT Δ_p is conformed to a parameterized XTG $\mathcal{X}_p$ if each run of $\mathcal{X}_p$ has a trace in Δ_P. The Theorem 4.5 is extended to the parameterized PDT with conditions of the fairness and the ordering relations. In addition to the implicit time-passing (formalized by the time-bounded node), the fairness condition prevents infinite stuttering. We require (see [18]) that transitions of the parameterized XTG corresponding to the stuttering in Δ_p, or time-passing do not increase the value of η such that $(\eta, \prec)$ is appeared as an ordering annotation on the outgoing edge of node.

Theorem 6.4. *Let* $\mathcal{X}_p$ *be a parameterized XTG* $\langle Init, L, \vec{l_0}, I, E, U\rangle$. *The parameterized PDT* $\Delta_p = \langle N, N_0, R_\mu, R_\tau, o, \Theta\rangle$ *over* L *and* $\mathcal{P}_Q$ *conforms to* $\mathcal{X}_p$ *if all of the following conditions hold:*

1. $Init \wedge I(\vec{l_0}) \Rightarrow \bigvee_{\langle \vec{l_0}, P\rangle \in N_0} P$

   In words, the conjunction of the global initial conditions of XTG and the set of invariants of the initial locations imply the predicates of the initial node of Δ_p *associated with the initial locations in* $\mathcal{X}_p$.

2. *For any node* $n = \langle \vec{l}, P\rangle$ *of* Δ_p:

(a) *For any edge* $e = \langle l, g, u, l' \rangle$ *of a process of* $\mathcal{X}_p$ *such that* $l \in \vec{l}$ *and* $sync(e, e') = \perp$ *for any edge* $e' \in E$, V_u *is the set of all variables* v *that are updated by* u *(i.e. such that* $\langle v, e \rangle \in u$ *for some* e*), and* N' *is the set of all nodes* $n' = \langle \vec{l'}, Q \rangle$ *of* Δ_p *such that* $\vec{l'}$ *is the result of* $\vec{l}$ *by the transition of process from* l *to* l'.

$$\begin{aligned} & P \wedge g \wedge I(\vec{l}) \wedge I'(\vec{l'}) \\ & \wedge \Big(\bigwedge_{\langle v,e \rangle \in u} v' = e \Big) \wedge \Big(\bigwedge_{v \in V \setminus V_u} v' = v \Big) \\ \Rightarrow \quad & \bigvee_{\langle \vec{l'}, Q \rangle \in N'} Q' \end{aligned}$$

For the local transitions from the processes which are not synchronized with any other transition, the label of node n*, the invariants associated with the active locations before and after the transition of* $\mathcal{X}_p$*, as well as the update generated by this one imply the predicates of some node in* N'.

(b) *two edges* $e_1 = \langle l_1, g_1, u_1, l_1' \rangle$ *and* $e_2 = \langle l_2, g_2, u_2, l_2' \rangle$ *are from two different processes in* $\mathcal{X}_p$ *such that* $l_1, l_2 \in vecl$ *and* $sync(e_1, e_2) \neq \perp$, V_u *is the set of all variables* v *that are updated by* u_1 *or* u_2 *such that* $(v, e) \in sync(e_1, e_2)$*, and* N' *is the set of all nodes* $n' = \langle \vec{l'}, Q \rangle$ *of* Δ_p *such that* $\vec{l'}$ *is the result of* $\vec{l}$ *by the joint-transition of two processes.*

$$\begin{aligned} & P \wedge g_1 \wedge g_2 \wedge I(\vec{l}) \wedge I'(\vec{l'}) \\ & \wedge \Big(\bigwedge_{\langle v,e \rangle \in u_1 \cup u_2} v' = e \Big) \wedge \Big(\bigwedge_{(v,e) \in sync(e_1,e_2)} v' = e \Big) \wedge \Big(\bigwedge_{v \in V \setminus V_u} v' = v \Big) \\ \Rightarrow \quad & \bigvee_{\langle \vec{l'}, Q \rangle \in N'} Q' \end{aligned}$$

For any pair of transitions which are synchronized, the label of node n *and the invariants of all active locations before and after the joint-transition, as well as the updates generated locally by each transition and the exchange of values between the processes imply the predicates of some node of* N'.

3. *For any node* $n = \langle \vec{l}, P \rangle$ *of* Δ_p*, the set of all nodes* N'' *such that* $n'' = \langle \vec{l}, Q \rangle$,

the same vector of control locations w.r.t n such that $n \rightarrow_\tau n''$.

$$\begin{aligned} & P \wedge \delta \in \mathbb{R}^{\geq 0} \wedge {}_{c \in V_c} c' = c + \delta \wedge \bigwedge_{v \in V \setminus V_c} v' = v \wedge I(\vec{l}) \wedge I'(\vec{l}) \\ & \wedge \forall \varepsilon \leq \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow\; I''(\vec{l}) \\ & \wedge \forall \varepsilon < \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \;\Rightarrow\; \bigwedge_{\langle \vec{l}, g, u, \vec{l}' \rangle \in U} \neg g'' \\ \Rightarrow \quad & \bigvee_{\langle \vec{l}, Q \rangle \in N''} Q' \end{aligned}$$

For any time-passing transition by amount δ *that does not activate any urgent transition of* $\mathcal{X}_p$ *and the predicate label of* n *and the invariants of all active locations before and after the time-passing transition (for all the intermediate values of time) imply that the label of some node of* N'' *is checked*

4. *For any node* $n = \langle \vec{l}, P \rangle$ *of* Δ_p*:*

   (a) *for any node* $n' = \langle \vec{l}', Q \rangle$ *of* Δ_p *with* $n \rightarrow_\mu n'$

      i. *for any edge* $e = \langle l, g, u, l' \rangle$ *from which the transition corresponds to the control location* $\vec{l}$ *and* $\vec{l}'$ *such that* $sync(e, e') = \perp$ *for any edge* $e' \in E$ *with the same notations under the condition (2a):*

$$\begin{aligned} & P \wedge g \wedge I(\vec{l}) \wedge I'(\vec{l}') \wedge Q' \\ & \wedge \left(\bigwedge_{\langle v, e \rangle \in u} v' = e \right) \wedge \left(\bigwedge_{v \in V \setminus V_u} v' = v \right) \\ \Rightarrow \quad & \bigwedge_{(\eta, \prec) \in o(n, n')} \eta' \prec \eta \end{aligned}$$

      ii. *for any pair of edge* $e_1 = \langle l_1, g_1, u_1, l_1' \rangle$ *and* $e_2 = \langle l_2, g_2, u_2, l_2' \rangle$ *under the condition (2b) and with the same notations:*

$$\begin{aligned} & P \wedge g_1 \wedge g_2 \wedge I(\vec{l}) \wedge I'(\vec{l}') \wedge Q' \\ & \wedge \left(\bigwedge_{\langle v, e \rangle \in u_1 \cup u_2} v' = e \right) \wedge \left(\bigwedge_{\langle v, e \rangle \in sync(e_1, e_2)} v' = e \right) \wedge \left(\bigwedge_{v \in V \setminus V_u} v' = v \right) \\ \Rightarrow \quad & \bigwedge_{(\eta, \prec) \in o(n, n')} \eta' \prec \eta \end{aligned}$$

   (b) *for any node* $n'' = \langle \vec{l}'', Q \rangle$ *of* Δ_p *with* $n \rightarrow_\tau n''$ *and any transition*

$$
\begin{aligned}
& n \rightarrow_{\mu} n' \ in \ \Delta_p : \\
& \qquad P \wedge \delta \in \mathbb{R}^{\geq 0} \wedge {}_{c \in V_c} c' = c + \delta \wedge \bigwedge_{v \in V \setminus V_c} v' = v \wedge I(\vec{l}) \wedge I'(\vec{l}'') \wedge Q' \\
& \qquad \wedge \forall \varepsilon \leq \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \ \Rightarrow \ I''(\vec{l}) \\
& \qquad \wedge \forall \varepsilon < \delta : \bigwedge_{c \in V_c} c'' = c + \varepsilon \wedge \bigwedge_{v \in V \setminus V_c} v'' = v \ \Rightarrow \bigwedge_{\langle \vec{l}, g, u, \vec{l}' \rangle \in U} \neg g'' \\
\Rightarrow \quad & \bigwedge_{(\eta, \prec) \in o(n, n')} \eta' \preceq \eta
\end{aligned}
$$

Proof. This theorem is a direct consequence of Theorem 4.5. We can use the proof of Theorem 4.5. The proof of conditions 1 - 3 of Theorem 6.4 are similar to the proof of conditions 1 - 3 of Theorem 4.5. For the proof of condition 4, we assume that $n \rightarrow_{\mu} n'$ and $n \rightarrow_{\tau} n''$ and that $(\eta, \prec) \in o(n, n')$ and $(\eta, \preceq) \in o(n, n'')$ respectively. By induction hypothesis and the choices of n' and $n \rightarrow_{\mu} n'$ and n'' and $n \rightarrow_{\tau} n''$, we have $(l_{ij}, l'_{ij}) \models \eta \prec \eta'$ and $(l_{ij}, l'_{ij}) \models \eta \preceq \eta'$ as required.

□

6.4 Verification of properties related to the Whole System

By using Fischer's mutual exclusion protocol as a running example, we show that verifying properties related to the entire system with Δ_p is confidently possible and successfully works. Let's recall the protocol specification in XTG. In fact we give a name to each transition, i.e. *try*, *set*, *abort*, *enter*, *exit*, among four control locations in order to describe its behaviours shown in Figure 6.2.

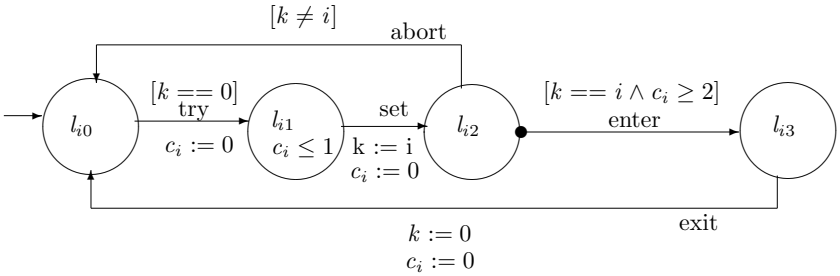


Figure 6.2: *XTG specification for Fischer's protocol*

For the N process version of Fischer's protocol, we write $p[i].loc$ and $p[i].c$ to denote the control location and the local clock of process i, and denote by cs the set of processes that are in their critical section, i.e.

$$cs \ = \ \{i \in 1, \ldots, N : p[i].loc = 3\}$$

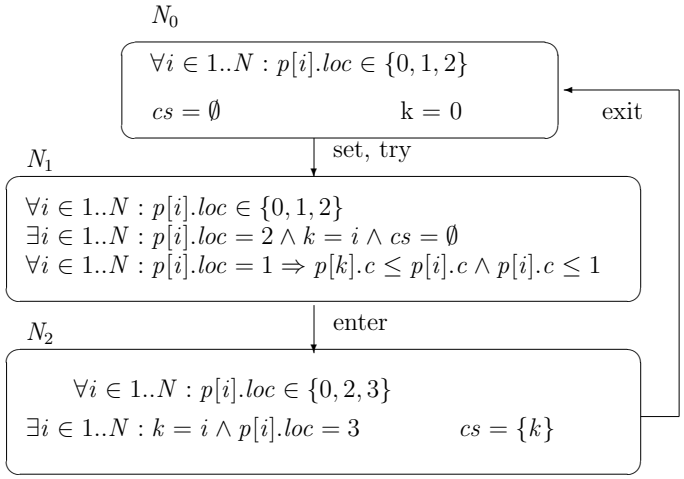


Figure 6.3: $\Delta 1_p$ *for Fischer's protocol for N processes*

As for the two-process version, the system contains a shared variable k that holds the process allowed to enter the critical section. The overall approach to verification proceeds in two steps as before by establishing conformance of Δ_p with respect to $\mathcal{X}_p$ and by model checking the finite state Δ_p.

Figure 6.3 gives $\Delta 1_p$ for Fischer's protocol for N processes, which can be shown to conform to the $\mathcal{X}_p$ by discharging a set of proof-obligations based on the conditions of Theorem 6.4 and proving them. Edges have names that represent corresponding transitions in $\mathcal{X}_p$. For example, here we consider the possible transitions from the node N_1 in the $\Delta 1_p$ of Figure 6.3.

Of particular interest are the transitions of process k: after a certain process i takes a transition from location 1 to location 2, the process i can be the process k with reset local clock ($c_i = 0$) in the fact that the process i sets k to its own id. Some other processes, which are in location 1 with clock constraint ($c \leq 1$), also wait for entering to location 2 in case their local clocks should be greater than equal to the process k's local clock since whenever the process i takes a transition from location 1 to location 2, its clock becomes reset and the other processes are still in location 1 as increasing their clocks. The corresponding predicate

$$\forall i \in 1, \ldots, N : p[i].loc = 1 \Rightarrow (p[k].c \leq p[i].c) \wedge (p[i].c \leq 1)$$

added into the node N_1

This process k has transition to control location 3 is covered by the transition

from node N_1 to node N_2 of the $\Delta 1_p$, because the corresponding proof obligation

$$\begin{array}{ll} & N_1 \wedge\ k' = k \wedge\ cs' = \{k\} \\ & \wedge\ p' = [p\ EXCEPT\ (p[i]'.loc = 3)] \\ \Rightarrow & N_1' \vee N_2' \end{array}$$

obviously holds.

The remaining proof obligations are similar, and CVC-LITE can be used to discharge them. Those proof obligations and CVC-LITE codes for the proof obligations are shown in Appendix A (Again, $\Delta 1_p$ of Figure 6.3 contains no time-passing edges other than the self-loops, which we do not show explicitly according to our convention.)

Regarding $\Delta 1_p$ as finite labelled transition systems, their runs can be encoded in the input language of standard model checkers. We add special variables to $\Delta 1_p$ can be used to prove the mutual exclusion property, which states that no two processes can be simultaneously inside their critical sections, which can be expressed as the LTL formula

$$\Box(\forall i, j \in 1, \ldots, N : i \in cs \wedge j \in cs \Rightarrow i = j).$$

In this case, it is not quite enough to consider the predicates as Boolean variables and use standard model checkers because we need elementary set theory to infer mutual exclusion from the node labels. Model checking would still work if extra predicates were added to the labels of $\Delta 1_p$ nodes.

In alternative way, we can use CVC-LITE to infer the mutual exclusion property, for example we check the following formula

$$cs = \emptyset \vee cs = \{k\}$$

on every node of $\Delta 1_p$ by the help of CVC-LITE.

6.5 Verification of Universal Properties

So far, we have successfully applied $\Delta 1_p$ in proving the mutual exclusion protocol. However the approach we used for verification of properties related to the entire system cannot be used to prove the individual accessibility of that protocol since it doesn't enable us to keep track the behaviors of some particular process. In general, this approach suffers from the limitation that it cannot be used for the verification of universal properties.

Many properties of parameterized systems are naturally expressed as formulas of the form $\forall i \in 1..N : P(i)$ where $P(i)$ is an LTL property that describes a property of a single process. Such properties can be verified by distinguishing a single process $i \in 1..N$ from the rest of the processes. Formally, this corresponds to introducing a Skolem constant i and proving the property $i \in 1..N \Rightarrow P(i)$.

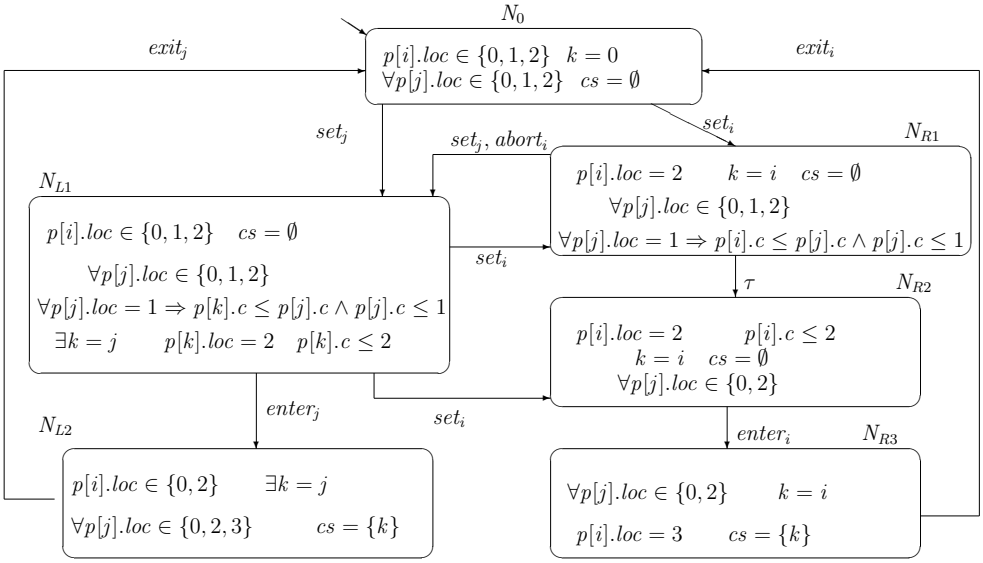


Figure 6.4: $\Delta 2_p$ *for Fischer's protocol for N processes.*

The idea in constructing Δ_p for a *universal* property is thus to follow the evolution of the distinguished process i and to represent the transitions of the other processes in a separate part of the predicate diagram. $\Delta 2_p$ in Figure 6.4 illustrates this idea for the N-process version of Fischer's protocol. In order to simplify the node labels, we write $\forall Q[j]$ as a short-hand for

$$\forall j \in 1, \ldots, N \setminus \{i\} : Q[j]$$

and similarly for $\exists Q[j]$.

In fact, we add N_{R1}, N_{R2}, and N_{L1} in $\Delta 2_p$ to represent particular interest of processes i and j are at location 2: first see N_{R1} and take a look at the downward edge to N_{R2}; after a certain process i takes a transition set_i from location 1 then process i stays at location 2 until the other processes group j enters to location 2.

Now let us consider the edge from N_{L1} to N_{R2} labelled as set_i, $abort_j$. After a certain process took the transition set_j (an edge from N_{L0} to N_{L1}), it becomes the process with k and resets its clock $c = 0$. The clocks of all other processes at location l_1 continue to increase their clocks. The corresponding predicate

$$\forall j \in 1, \ldots, N : p[j].loc = 1 \Rightarrow (p[k].c \leq p[j].c) \wedge (p[j].c \leq 1)$$

added into the node N_{L1}.

While the process k waits for entering to l_3, the process i can arrive at l_2 and become the gaining process (i.e., enregister i to k). This transition corresponds to the edge from N_{L1} to N_{R1} or N_{R2} according to the fact that there are other processes at the location l_1 or not. In particular, node N_{R2} represents the "change of the gaining process".

For the conformance checking, we consider the possible transitions of process j from the node N_{L2} of Figure 6.4 with corresponding control locations: the transition to location 3 ($enter_j$) is captured by the downward edge, and indeed the proof obligation

$$\begin{array}{ll} & N_{L1} \wedge \ k' = k \wedge \ cs' = \{k\} \\ & \wedge \ p' = [p \ EXCEPT \ (p[j]'.loc = 3)] \\ \Rightarrow & N'_{L2} \vee N'_{L1} \end{array}$$

is easily seen to hold (and proven by CVC-LITE). Note the complete proof is done in the Appendix A in order to prove that the $\Delta 2_p$ is a correct abstraction of the N-process version of Fischer's protocol. Again, we use Theorem 6.4 for proving this conformance.

The diagram $\Delta 2_p$ enables us to show the liveness property of the protocol that one of the processes will end up eaching the critical section, i.e. LTL formula will be checked

$$\forall i \in 1, \ldots, N : \Box\big((k = i \wedge p[k].loc = 2) \Rightarrow \Diamond(p[k].loc = 3)\big)$$

Also, the mutual exclusion property of Fischer's protocol for N-process can be verified using $\Delta 2_p$ in Fig 6.4 in a similar way: we can again use CVC-LITE to infer the alternative formula

$$\forall i \in 1, \ldots, N : cs = \emptyset \vee cs = \{k\}$$

by checking it over every node in $\Delta 2_p$. Thus, in general we can say that Δ_p can be used to prove the universal properties of $\mathcal{X}_p$, and verify the properties related to the whole system.

6.6 Conclusion

In this chapter we have defined parameterized XTGs ($\mathcal{X}_p$) and PDTs (Δ_p). By using the N-process version of Fischer's protocol as a running example, we have shown that with a little extention/modification on the conformance Theorem of ordinary PDT, it is possible to apply Δ_p to the verification of parameterized systems specified in $\mathcal{X}_p$.

More specifically, in model checking perspective, our particular interest was to handle limitations on the data model which stem from the verification tools (such as PMC, UPPAAL) to which the XTG or other timed specifications are input.

For such an application, a family of processes was represented in a single diagram in order to verify properties related to whole processes with Δ_p.

We were also interested in universal properties of parametric systems, and they could be established by distinguishing a single process and following its behavior separate from that of the remainder of the system.

Chapter 7

Case Study: Generation PDTs and Verification

We have seen how our methodology can be seamlessly and securely extended to support new verification techniques based on the idea that an abstract model can be used in place of a more complex model for the purpose of doing efficient verification. However, we have yet to exhibit how the theorem proving component can help with the verification in various ways, and also that the model checking component, though in its infancy, can handle more than just Fischer mutual exclusion examples.

In this chapter we use a case study to demonstrate how the use of interactive theorem proving enables us to generate a complete abstract representation (PDT), and to verify some safety property over the PDT as a running example railway crossing system. Then this case study is extended as verifying a parameterized railway crossing system. We first explain about the railway crossing system in the following section.

7.1 Railway crossing system

Figure 7.1 depicts a railway crossing system involving a gate, and two sensors *approach* and *leave* respectively telling that a train is approaching and leaving the passage way. This is a classical example of a real-time system assuming that there is only a single train and the gate. We will propose richer models for this example such as the railway crossing lies in a region of interest $\mathbf{R}$ in multiple tracks and a number of trains travel through the $\mathbf{R}$.

The system can be modelled in three XTGs described in Figure 7.2 as a train process, a gate controller, and a gate. (Note: black circle marks on some locations indicate urgent conditions mentioned in Definition 3.5). The integer variable *cnt*

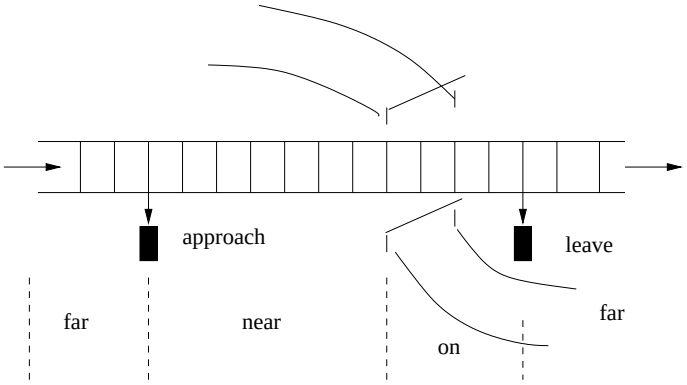


Figure 7.1: *A railway crossing system*

in the controller process is used to count the number of trains moving between the signal *approach* and *leaving*. Then the current system can easily be extended to any number of trains. The train process consists of three locations namely:

- *far* : trains are far from the *crossing*,
- *near* : trains are near to the *crossing*, and
- *on* : trains are on the *crossing*,

In case the number of sensor processes can be interpreted as the number of train processes. The following sections show the XTG specification of the system first, and then the PDT derived from the XTG. During the PDT derivation, we show how to discover new nodes and identify required abstraction-predicates. Note that in this example, abstraction-predicates are denoted as labels in the new nodes of PDT.

7.1.1 XTGs Design

The description of system is first translated into XTG's; When a train approaches to the near sensor, it issues a signal to the controller via the synchronization channel *approach* that means it will enter the *crossing* in six time units. Four time units later it leaves the *crossing*. This is also communicated to the controller process through the channel *leave*. Once a train is at the location *far*, it will enter the location *near* sometime between 5 and 20 time units.

The gate has four possible locations. Initially it is at the location *open*. When receiving a close signal, it transfers to the location *closing*. One time unit is allowed to close the gate enforced by [gc $\geq$ 1] clock constraint. Afterwards the gate becomes close and it waits an open signal at the location *close*. With an open

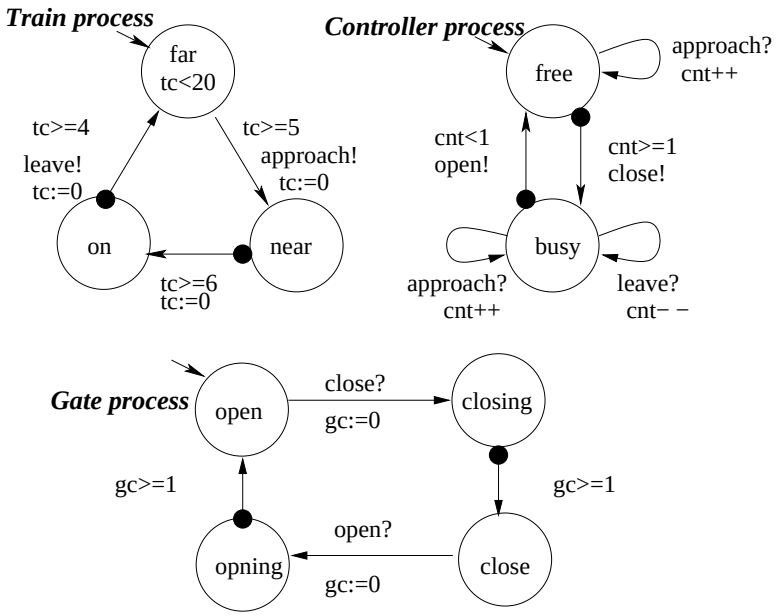


Figure 7.2: *Three XTG processes representing the system: Sensor, Controller and Gate*

signal it transfers to the location *opening*. One more time unit later it becomes open again.

The gate controller acts in response to the moving trains, by closing and opening the gate when appropriate. It counts the number of trains which are at location *near* or *on*. If no trains are near but one becomes near, the controller issues a close signal. If the last train left the *crossing*, an open signal is issued.

The safety property to be verified can be mentioned as "when a train is on the crossing, the gate is close". To formalize this safety property, we first assume that the set of atomic propositions

$$AP = \{train_i@on, \quad gate@close \mid 0 < i \leq N\}$$

where $train_i@on$ means that a process of train i is at the location *on* and $gate@close$ means that a process of gate controller is at the location *close*. We will use i as process identities. N is the number of processes.

In the following formulation, we use universal and existential quantifications over the set of identities. Strictly speaking, these are not part of LTL. Since we deal with a finite number of processes the universal quantification $\forall i.P(i)$, where P is some proposition on processes, can be expanded into $P(1) \wedge ... \wedge P(N)$, and similarly we can expand $\exists i.P(i)$. The quantifications can thus be considered as simply abbreviations. The safety property is expressed now by below formula

$$\Box(train_i@on \Rightarrow gate@close) \tag{7.1}$$

and this verification problem will be further discusses in the following sections.

7.2 Generation PDTs and Verification

This section explains how to generate a complete PDT (Δ) semi-automatically. We use the term semi-automatically, since during the generation the user's intervention is still needed in particular for infering/identifying necessary predicates based on verification conditions of Theorem 6.4, which is extended version of Theorem 4.5 by regarding synchronizations related to $\mathcal{X}$, and rewriting those conditions in CVC-LITE formulas to prove its validity by conformance checking w.r.t XTGs ($\mathcal{X}$).

As importantly, our main concern is to prevent constructing incomplete Δ in that sense some predicates inferred by human may mislead users into generating a Δ, which does not conform to $\mathcal{X}$. Our tendency now is attempt to find a way to identify required predicates that help users to avoid building such a Δ inconformed to $\mathcal{X}$.

According to IAR, a skeletal abstract representation is extracted syntactically from $\mathcal{X}$ specification (ref. Backward Process) and iterative refinement is required to complete this skeletal model as checking its conformance w.r.t $\mathcal{X}$. The user can infer predicates (labels) of new nodes (post nodes of current one), which can be found by analyzing a set of verification conditions discharged by Theorem 4.5. Those conditions can be rewritten in CVC-LITE formulas and verified their validity. Although user interaction is sill necessary in this mechanism such as diagnosing (failed) proofs, this method can improve automation both in proving the verification conditions by using CVC-LITE and pointing to necessary refinements in case of failure. The example with the railway crossing system given in this section shows how our approach works.

In order to derive Δ for the railway system specifying in $\mathcal{X}$, we assume this system consists of two trains for the simplicity.

Skeletal-abstraction. We first generate the "skeletal" abstract representation Δ_1 from $\mathcal{X}$ in Figure 7.3; the system may be viewed as a protocol controlling the gate movement towards train's access to the *crossing* and this gate controller is modelled very simply in which it cycles through an infinite sequence of **free** and **busy**. In terms of system's protocol, we know the permission for movement of gate process is regulated by sensors detecting the passing trains, i.e. the gate process knows when it allows to open its gate or close according to the sensors. Thus, the predicate diagram Δ_1, which has a set of nodes containing labels as considering behaviours of controller process $controller = \{free, busy\}$ and train process $train = \{far, near, on\}$, is obtained.

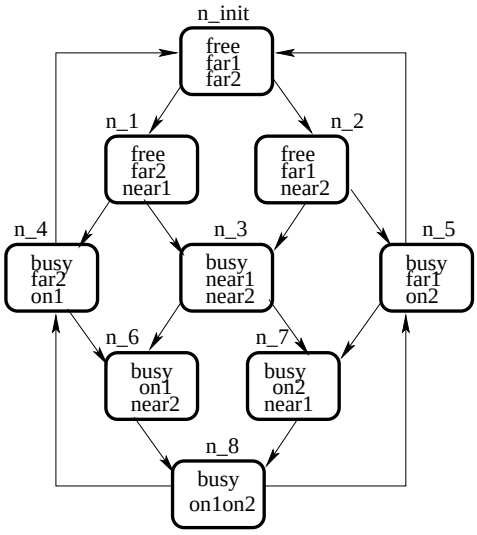


Figure 7.3: Δ_1*: Railway crossing model: initial skeletal predicate diagram*

However, according to our IAR, Δ_1 is still incomplete in a sense that the safety formula 7.1 is not decidable over the Δ_1 since Δ_1 doesn't have any node contains labels for atomic formulas that occur in the property, i.e. Δ_1 is missing labels in terms of information of a gate process. We consider those missing labels as required predicates should be added in Δ_1 that help to construct a complete predicate diagram.

We have inferred a set of necessary predicates, which was indicated by seeing undecidability of property over Δ_1. Now we attempt to refine Δ_1 by adding the predicates such as $gate = \{open, closing, close, opening\}$ that represent possible locations of gate process. Interesting is that as considering transitions of two train processes, we see each process takes a transition to possible successor locations by its local clock $(t1c, t2c)$. Moreover a controller process changes its state by counting a number of trains (cnt), which are either location at *near* or *on*. As a result, Δ_1 is refined regarding those important predicates we have inferred and shown as Δ_2 in Figure 7.4. Nodes with thick frames describe the mapping nodes of Δ_1 in Figure 7.3.

Now we ask two questions: (1) is the safety property decidable over this Δ_2? (2) is this Δ_2 conformed to the $\mathcal{X}$?

The verification problem in terms of the first question can be supported by two possible ways: (a) using the DIXIT toolkit or any model checker; (b) using CVC-LITE again to infer the property formula from each of the node labels of Δ_2. In (b) case we need alternative formula as follows:

$$(On1 \vee On2) \Rightarrow close$$

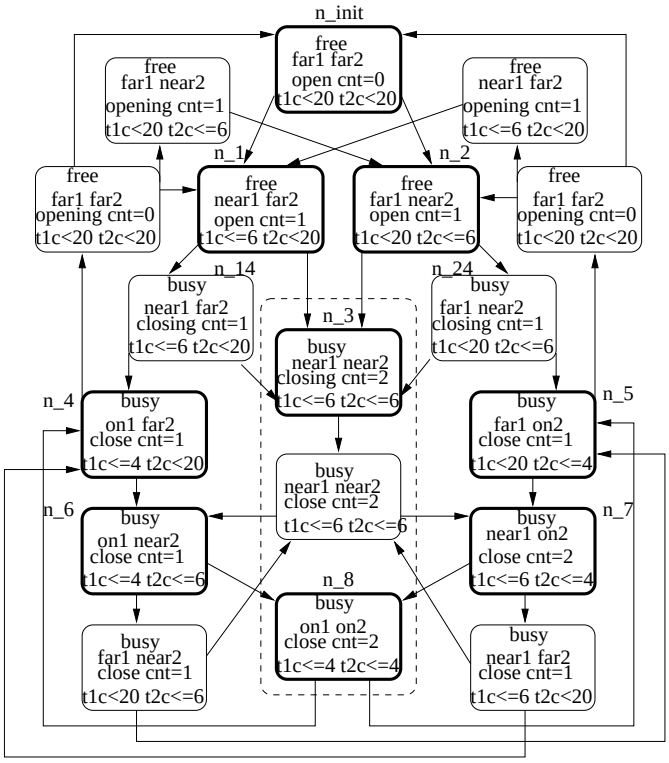


Figure 7.4: Δ_2*: Railway crossing model: adding events of Trains and Gate processes*

and CVC-LITE queries this alternative formula over every node and returns "Valid" as the result of verification. Both verification method-(a) and (b) successfully prove that the given safety property is valid over Δ_2.

As considering the second question, nevertheless we see that the safety formula 7.1 is decidable over Δ_2, this Δ_2 doesn't seem to be correctly conformed to $\mathcal{X}$ since some information is still missing. If we are only interested in verifying the property 7.1 then Δ_2 is enough to satisfy our interest. However, our true goal is to generate a complete abstract representation which not only preserves the property but also correctly conforms to $\mathcal{X}$. Thus we know another refinement has to be applied to Δ_2.

For instance, each train approaches to the *crossing* section and leaves it with different speed. A gate controller keeps the gate closed if any sensor detects a train passing by and the controller-process also needs a new label *gc* for its local clock. In order to express more information about which train precedes over the other one, Δ_2 needs new labels such as $t1c \geq t2c$ or $t2c \geq t1c$, that describe different speeds of two trains moving among $far, near, on$.

Those new labels can be added in Δ_2, and would cause splitting some nodes n_3 into n_{31} and n_{32} in which bring duplicated dashed edges in Figure 7.5. Such a set of extra dashed edges by splitting has to be queried its validity with help of CVC-LITE based on a set of verification conditions of Theorem 4.5. CVC-LITE returns "Invalid" for the queries of dashed edges. Therefore the set of dashed edges are to be removed from Δ_2.

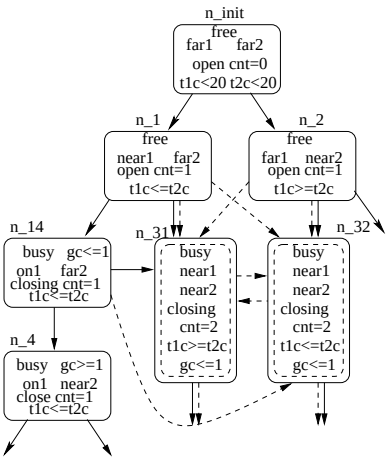


Figure 7.5: *The partially refined model of* Δ_2

As continuing our approach, a set of nodes of Δ_2 grouped by a dash-line in Figure 7.4 are split into two sub-nodes which are also grouped by a dash-line in Figure 7.6. It is easy to establish conditions $1 - 3$ of Definition 4.6 in order to prove that Δ_2 is structurally refined by Δ_3 because Δ_3 allows fewer transitions than the former. Finally CVC-LITE can generate proof obligations that ensure Δ_3 conforms $\mathcal{X}$. Note the complete proof is done by reproducing CVC-LITE input in the Appendix B.

CVC-LITE queries about the validity of the alternative formula over every node in Δ_3 and it returns "Valid". Therefore we can say that Δ_3 in Figure 7.6 inherits the desired safety property and finally we ensure that Δ_3 is the complete abstract representation of $\mathcal{X}$.

7.3 Verification of the parameterized railway crossing system

For the N-process version of railway crossing system, we write $t[i].loc$ and $t[i].c$ such that $t[i].loc \in \{far, near, on\}$ and $0 \leq t[i].c < 20$ in order to denote the control location and the local clock of train process i respectively. Also several notations for other processes are denoted as following:

- controller process: $ctl.loc \in \{free, busy\}$

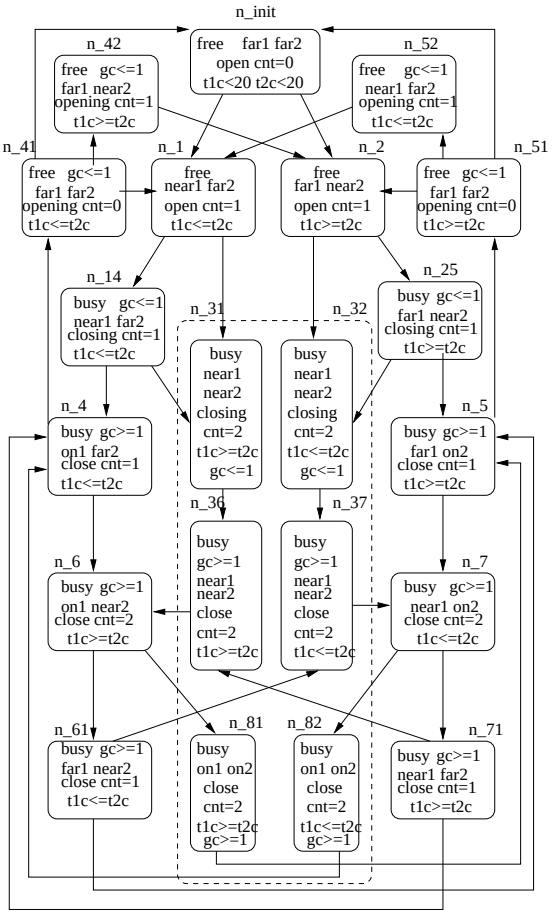


Figure 7.6: Δ_3*: Railway crossing model: approaching-time based implementation*

- counter of controller process: $ctl.cnt \in \{0, 1, .., N\}$
- gate process: $g.loc \in \{open, closing, opening, close\}$
- local clock of gate process: $g.c$

The overall approach to the verification problem is done in two steps as we have shown before – by establishing conformance of Δ_p w.r.t. $\mathcal{X}_p$, and – by model checking Δ_p.

As mentioned, in order to handle verification of universal properties for the parameterized railway crossing system, we distinguish a single process $i \in 1..N$ from the rest of the processes and represent the transitions of the other processes

in a separate part of the predicate diagrams. It can be noted as the node labels below

$$\forall j \in 1..N \setminus \{i\} : t[j]$$

Simply we have no mutex property here: there is no rule for limiting the number of train-processes which enter the "crossing" section and stay there, i.e. it can be possible that more than one process exist in the "crossing" section at the same time. With this railway crossing system, rather than counting/controlling the number of train processes in the "crossing", more highly required safety property is that when trains are in the "crossing" section, the gate should be closed no matter how many trains are in the crossing and it can be expressed by the LTL formula:

$$\forall i \in 1...N : \Box(t[i].loc = on \Rightarrow g.loc = close) \tag{7.2}$$

Figure 7.7 gives Δ_p for verifying universal properties for the N-process that can be shown to conform to $\mathcal{X}_p$ (cf. Figure 4.7) by discharging conditions of Theorem 6.4. A particulary interesting fact with this system is that kind of "pattern-of-change-value" for local-clocks of processes towards before after transitions of trains: after a certain process i takes a transition from any location to its successor location, the process i's local clock $t[i].c$ becomes zero because of the reset local clock condition ($tc := 0$). Some other processes $t[j]$, which do not take transitions yet in case their local clocks $t[j].c$ should be greater than equal to the process i's local clock.

For example, we consider one possible condition for a transition of process i ($t[i]$) from location *far* to *near* regarding its clock constraint condition (cf. $5 \leq tc \leq 20$); a train cannot be stopped instantly and restarting also takes time. Therefore, there are timing constraints on the trains before entering the *near*. If along all the rest of other processes are still at location *far* in case the gate controller process is at location either *free* or *busy* and the gate process closing its gate bar towards controller's state. The *ctl.cnt* in the controller process that counts the number of trains moving to between location *near* and *on*, and sets its value to 1.

Suppose this condition is true then the local clock of process i ($t[i].c$) is less than equal to the other processes local clocks since $t[i].c$ becomes reset and the other processes are still at location *far* increase their local clocks until they satisfy the same clock constraint condition in the sense that they will be at location *near* sometime between 5 and 20 time units. The corresponding predicates

$$\begin{aligned}
&\forall t[j].loc \in \{far, near, on\} \\
&\wedge \;\; ctl.loc \in \{free, busy\} \wedge ctl.cnt \in \{0, \ldots, (N-1)\} \\
&\wedge \;\; g.loc \in \{open, closing\} \wedge t[i].loc = near \\
&\wedge \;\; \big(\forall t[j].loc = far \wedge ctl.cnt = 1 \wedge ctl.loc = free \wedge g.loc = open \\
&\qquad \Rightarrow \;\; t[i].c \leq t[j].c\big)
\end{aligned}$$

added to node n_{L1} as its label. The predicates for n_{R1} is symmetrical.

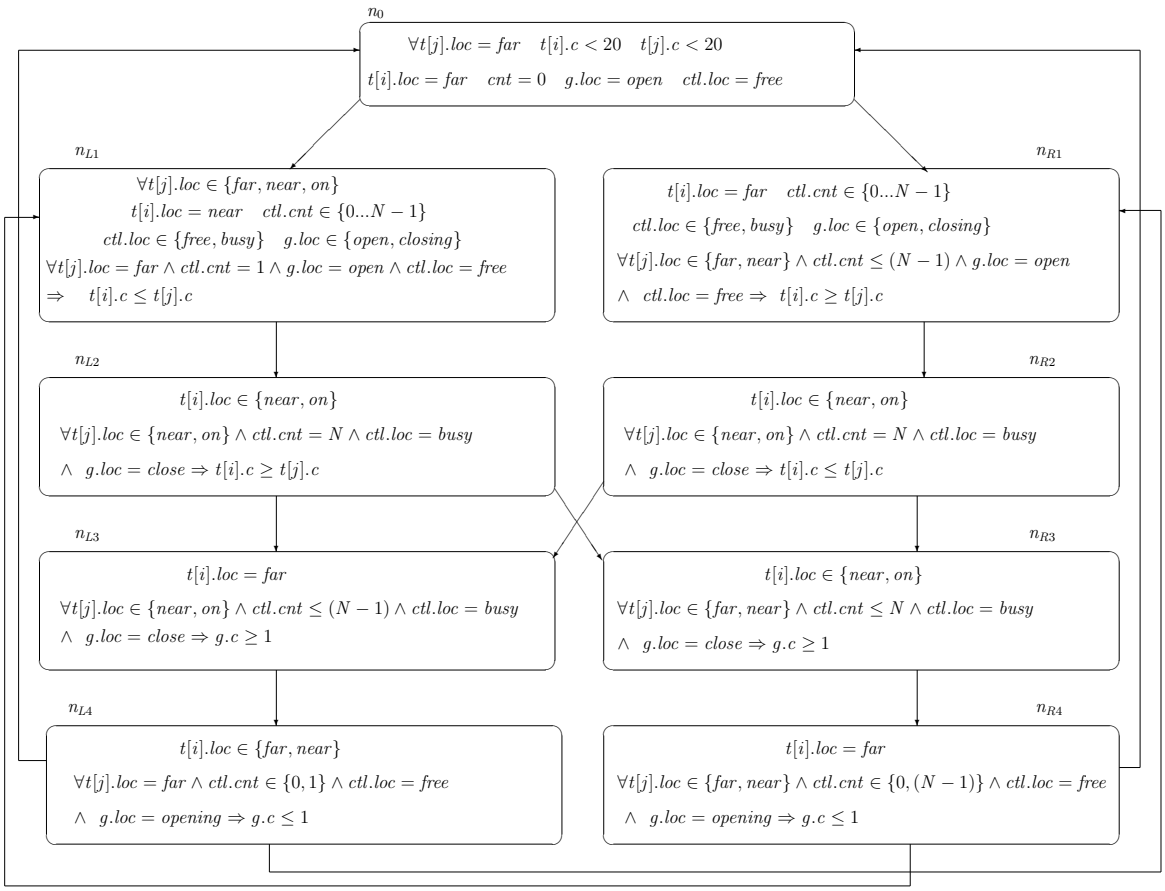


Figure 7.7: Δ_p *: Railroad crossing model with N-processes*

The predicates in node n_{L2} (n_{R2} is symmetrical) shows that $t[i]$, resets its local clock in order to reach *on* or it just stays at *near*. Either the train-i is at *on* or *near*, a controller waits for a signal from the $t[i]$. After the controller receives a signal, the controller sends another signal to a gate, and sets itself *busy* as counting the number of trains are at *near* or *on*. In the meantime, we consider that other trains ($t[j]$) are either at *near* or *on* in case their local clocks become less than the local clock of $t[i]$ for the similar reason as explained before. Thus, node n_{L2} has such conditions as its predicate in it.

When $t[i]$ is at location *on* and the other train group $t[j]$ is at *far* or *near* in case the controller is at location *busy* with *close*! synchronization for the gate process, where the counter has value $N-1$ or N regarding the location $t[j]$ then the local clock $t[i].c$ becomes less than equal to $t[j].c$. This condition can be

explained by predicates labels in n_{R3}.

Some conditions in terms of the case that $t[i]$ still stays at *far* and other $t[j]$ are are either *near* or *on*, can be described by labels in n_{L3}.

Now $t[i]$ has crossed the passage way and sends a *leave*! signal. The controller can now let the gate be *opening* from *close* with a *open*! signal and sets itself to *free*. In case all other trains $t[j]$ are either at *far* or *near*, the counter become zero or $N-1$ such that $ctl.cnt = 0$ when $t[j]$ group is all at *far* or $ctl.cnt = N-1$ when $t[j]$ is at location *near*. Since $t[i]$ already left the passage way $ctl.cnt$ is no longer to be equal to N, and the corresponding predicates are added in node n_{R4}.

Now $t[j]$ group is approaching to *near* and $t[i]$ goes far from the region of interest R. The remaining set of predicates of nodes can be explained in similar ways, and proofs are shown in detail in Appendix B with CVC-Lite code.

Upon building Δ_p, the required step coming along with Δ_p is checking the conformance w.r.t its $\mathcal{X}_p$ by discharging the conditions of Theorem 6.4. Note that required proof obligations are shown/proved as CVC-Lite code in Appendix B.

We have carried out work for extending the usage of Δ_p for the verification of parameterized railway crossing system so far: first we checked its conformance w.r.t $\mathcal{X}_p$. Now we move on next work such as establishing some property in which we are interested to verify so that the safety property shown as the LTL formula 7.1 is queried over every node of Δ_p.

7.4 Conclusion

In this chapter we have shown a case study that demonstrates the use of IAR. The example of railway crossing system showed how to generate a complete abstract representation (PDT) from a given system based on Theorem 4.5: First we have a skeletal abstraction from XTG. Then from the skeletal PDT's initial node and a set of configurations of initial control location mapped to the initial node, we inferred required predicates that help to generate post nodes as analyzing proof obligations obtained by Theorem 4.5.

Finally we prove that this PDT generated is a complete abstract representation in proving its conformance w.r.t. XTG and showing that safety property is successfully verified over itself.

For handling the parameterized railway crossing system, we applied same approach for verification of N-process proposed in chapter 6.

Chapter 8

Conclusions and Future works

8.1 Conclusions

This thesis presented a formal verification technique for real-time systems based on combination between different techniques such as predicate abstraction, theorem proving and model checking in a way that we presented an IAR, which is an integration methodology with HOL of a predicate abstract representation reduction technique that draws upon both SAT-, and model checking-based technologies.

It was developed by subsequently considering three key aspects identified in the introduction: abstract representation, abstract refinement and evaluation of the finite state abstraction. For each aspect, a solution was developed, with the basic assumption that abstract representation aspect is based on an abstraction manner, which reduces the number of states of the given concrete model as abstracting away behaviours not essential to the verification using IAR.

The described IAR approach in the evaluation perspective was aimed at verification of temporal logic properties, LTL, on a correct abstraction which is a conformed abstract-representation of an extension of timed automata.

At the abstract representation/refinement levels, given information such as a set of abstraction predicates, verification conditions and the concrete system description, we first computed (constructed) an approximate abstract representation (skeletal abstract representation) manually. Then, the skeletal abstract representation was checked its conformance w.r.t. the concrete system and the skeletal one refined appropriately; if conformance checking returns "incorrect" then repeated applying refinements until the the abstract representation becomes "correct".

In the meantime IAR also considers if properties of interest are decidable over the abstract representation in case both conditions (conformance and decidability)

are satisfiable with the abstract representation then we could say that the abstract representation is a complete abstract representation and LTL properties hold on the abstract model also hold on the concrete system. Otherwise, the abstract model is not complete. Therefore, we need to iteratively concretize (refine) the abstract model until it becomes complete. IAR terminates its prodecure with cases that verification condition for correct conformance valid and IAR can find no more new nodes and edges would be added into the abstract representation being considered.

Besides better illustration/implementation of IAR procedure, we proposed the format of predicate diagrams for timed systems (PDT) as a notation to represent Boolean abstractions of real-time systems. This format is a variant of predicate diagrams for discrete systems; in particular, time-passing transitions are distinguished from discrete ones. We also established a set of proof obligations for proving conformance between an XTG model of a timed system and an abstract representation based on conditions of Theorem 4.5.

Also these proof obligations, which are first-order formulas discharged by Theorem 4.5 conditions, can be applied in finding the right abstract representation and identifying the predicates for the valid abstraction based on analyzing verification conditions which are obtained during the IAR procedure. The set of proof obligations can be established using automatic provers for first-order logic, such as SMT solvers, CVC-LITE.

At the evaluation level, the LTL decision problem was reexpressed by Boolean satisfiability problem (SAT) in such a way the LTL property could be assigned in alternative formula to make the formula evaluate to TRUE. Equally important is to do determine that no such assignments exist, implying that the property expressed by the alternative formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the property is unsatisfiable over PDTs; otherwise it is satisfiable. CVC-Lite was queried about the satisfiability of this alternative formula.

In this all sense, PDTs constitute an interface between verification techniques based on deduction and model checking. Predicates are interpreted as labels of PDTs' nodes and considered as atomic propositions for model checking. The format of predicate diagrams is supported by the DIXIT toolkit, and we demonstrated its use regarding IAR spirits via Fischer's mutual exclusion protocol and a railway crossing system and also showed that our approach with PDTs can verify not only safety properties but also liveness properties.

We showed that the PDT format can equally well be used for parametric systems, such as the N process version of Fischer's protocol in Chapter 6. For these applications, a family of processes is represented in a single diagram. In fact, we are often interested in *universal* properties of parametric systems, and they can be established by distinguishing a single process and following its behavior separate from that of the remainder of the system.

Finally, we used a case study to fully demonstrate IAR enables us to generate a complete PDTs as running an example railway crossing system. Especially, to

emphasize the correctness of derivation PDTs, we used theorem proving iteratively during the PDTs construction. This case study was also extended to handle the parameterized ralway cross system.

8.2 Future works

We consider this work as a first step towards the application of Boolean abstractions in the verification of real-time systems. One of the current limitations lies in the fact that we abstract from the precise amount of time that may elapse in a time-passing transition. Thus, we cannot easily verify quantitative properties, such as upper bounds on global response times, although properties that mention individual clocks can be verified. We intend to study two possible solutions to this problem, either by using a timed temporal logic (TLTL) or by introducing auxiliary clocks during verification, as suggested by Henzinger et al. [34] and by Tripakis [57]. This would in particular allow us to take advantage of model checking tools for real-time systems such as Uppaal or PMC.

Besides, we aim at reducing the number of verification conditions that users have to discharge with the help of a theorem prover in order to establish conformance. In fact, we consider the proof obligations of Theorem 3.1 and Theorem 5.1 mainly as a litmus test to establish the conditions that a PDT should satisfy, and we observe that most of them are quite trivial for typical examples. It will be interesting to restrict attention to specific classes of systems that give rise to decidable proof obligations, thus enabling the automatic construction of PDTs.

We also intend to study in more detail techniques of refinement for the construction of PDTs, given an XTG and a set of predicates of interest. Preliminary work on combining tools for abstract interpretation and state space exploration has been reported in [38, 39], but more experience will be necessary in order to identify complete abstractions for real-time systems.

Although, the predicate abstraction of timed systems were extended to apply to richer models such as parameterized timed automata and even to timed automata with other infinite type such as counters or stacks, the price to pay of course it that such extensions are necessarily incomplete. In future work we would also like to address time-abstracting formulas with arithmetic and other constraints instead of only supporting propositional variables. In this way we could express and verify further interesting properties such as bounded response.

In terms of incompleteness of a SMT solver for some theories in that CVC-LITE is not capable of handling large and propositionally complex formulas in a rich combination of theories, especially supporting many numbers of quantifier instantiations, we could apply another theorem provers to manipulate such incompleteness of CVC-LITE. Nevertheless applying several different types of SMT solvers still gives additional burden on the users to learn the commands and tactics of the provers, or on automatic provers whose theory domains might not be rich enough to include the mathematics/theories that users attempt to prove.

Ideally one would like to have dedicated tools which are to support propositionally complex formulas in theories perhaps with limited guidance from users, and improve the automation of interactive provers by integrating them with automatic provers.

Appendix A

CVC-Lite Code of Fischer's Protocol

This appendix contains the comeplete proof of conformance between XTGs specification of Fisher protocol and PDTs by reproducing CVC-Lite input.

For the 2-process version of Fischer's protocol

```
process: TYPE = [#id:INT, loc: INT, k: INT, cs : INT, c: REAL #];

p1, p2: process;

pos: REAL -> REAL = LAMBDA (x:REAL): IF x>=0 THEN x
                                     ELSE (-x)   ENDIF;

%possible locations for processes

init_1 : BOOLEAN =
            p1.id = 1 AND p1.loc = 0 AND pos(p1.c) >= 0
            AND p1.cs = 0 AND p1.k = 0  ;

set_1 : BOOLEAN =
            p1.id = 1 AND p1.loc = 1 AND p1.cs = 0
            AND p1.k=0 AND pos(p1.c)<= 1 ;


try_1 : BOOLEAN =
            p1.id = 1 AND p1.loc = 2 AND pos(p1.c) < 2
            AND p1.k = 1 AND p1.cs = 0 ;
```

```
enter_1 : BOOLEAN =
             p1.id = 1  AND p1.loc = 3 AND pos(p1.c) >= 2
             AND p1.k = 1 AND p1.cs = 1 ;


init_2 : BOOLEAN =
            p2.id = 2 AND p2.loc = 0 AND pos(p2.c) >= 0
            AND p2.cs=0 AND p2.k=0 ;

set_2 : BOOLEAN =
            p2.id = 2 AND p2.loc = 1 AND p2.cs = 0
            AND p2.k=0 AND pos(p2.c)<= 1 ;


try_2 : BOOLEAN =
            p2.id = 2 AND p2.loc = 2 AND pos(p2.c) < 2
            AND p2.k = 2 AND p2.cs = 0 ;


enter_2 : BOOLEAN =
             p2.id = 2  AND p2.loc = 3 AND pos(p2.c) >= 2
             AND p2.k = 2 AND p2.cs = 2 ;

% possible transitions

init1_set1 : BOOLEAN = p1.k = 0  AND p1.c = 0;

set1_try1 : BOOLEAN = p1.k = 1 AND p1.c = 0 ;

try1_init1 : BOOLEAN = p1.k /= 1 ;

try1_enter1 : BOOLEAN = p1.k = 1 AND pos(p1.c) >= 2;

enter1_init1 : BOOLEAN = p1.k = 0 AND p1.c = 0 ;


init2_set2 : BOOLEAN = p2.k = 0  AND p2.c = 0;

set2_try2 : BOOLEAN = p2.k = 2 AND p2.c = 0 ;

try2_init2 : BOOLEAN = p2.k /= 2 ;

try2_enter2 : BOOLEAN = p2.k = 2 AND pos(p2.c) >= 2;
```

```
enter2_init2 : BOOLEAN = p2.k = 0 AND p2.c = 0 ;


% predicates of set of nodes

n1020 : BOOLEAN =  p1.id = 1  AND p2.id = 2
         AND p1.loc = 0 AND p2.loc = 0 AND p1.k = 0  AND p2.k = 0 ;


n1120 : BOOLEAN =  p1.id = 1 AND p2.id = 2
         AND p1.loc = 1 AND p2.loc = 0 AND p1.k = 0 AND pos(p1.c) <= 1;


nL1220 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 2 AND p2.loc = 0 AND p1.k = 1 AND p2.k = 1;


n1320 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 3 AND p2.loc = 0 AND p1.k = 1 AND p2.k = 1;

n1121 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 1 AND p2.loc = 1 AND p1.k = 0 AND p2.k = 0
         AND pos(p1.c) <= 1 AND pos(p2.c) <= 1;

nL1221 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 2 AND p2.loc = 1 AND p1.k = 1 AND p2.k = 1
         AND pos(p1.c) <= 1 AND pos(p1.c) <= pos(p2.c);

nL1222 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 2 AND p2.loc = 2 AND p1.k = 1 AND p2.k = 1
         AND pos(p1.c) <= pos(p2.c);

n1322 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 3 AND p2.loc = 3 AND p1.k = 1 AND p2.k = 1;

nL1022 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 0 AND p2.loc = 2 AND p1.k = 0 AND p2.k = 0;

nL1122 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 1 AND p2.loc = 2 AND p1.k = 0 AND p2.k = 0
         AND pos(p1.c) <= 1;
%

n1021 : BOOLEAN =  p1.id = 1 AND p2.id = 2 AND p1.loc = 0
```

```
         AND p2.loc = 1 AND p1.k = 0 AND pos(p2.c) <= 1;

nR1022 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 0 AND p2.loc = 2 AND p1.k = 2 AND p2.k = 2;

n1023 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 0 AND p2.loc = 3 AND p1.k = 2 AND p2.k = 2;

nR1122 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 1 AND p2.loc = 2 AND p1.k = 2 AND p2.k = 2
         AND pos(p1.c) <= 1 AND pos(p2.c) <= pos(p1.c);

nR1222 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 2 AND p2.loc = 2 AND p1.k = 2 AND p2.k = 2
         AND pos(p2.c) <= pos(p1.c);

n1223 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 2 AND p2.loc = 3 AND p1.k = 2 AND p2.k = 2;

nR1220 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 2 AND p2.loc = 0 AND p1.k = 0 AND p2.k = 0;

nR1221 : BOOLEAN = p1.id = 1 AND p2.id = 2
         AND p1.loc = 2 AND p2.loc = 1 AND p1.k = 0 AND p2.k = 0
         AND pos(p2.c) <= 1;

% init
PUSH;
ASSERT init_1 AND init_2;

QUERY n1020;
POP;

% n1020 --> n1120
PUSH;
ASSERT n1020 AND init_1 AND init1_set1 AND set_1 AND init_2;

QUERY p1.k = 0 AND p2.k = 0 AND pos(p1.c) <= 1  ;
POP;

% n1120 --> n1121
PUSH;
ASSERT n1120 AND set_1 AND init_2 AND init2_set2 AND set_2;
```

```
QUERY n1121
%QUERY p1.k = 0 AND p2.k = 0 AND pos(p1.c) <= 1 AND pos(p2.c) <= 1;
POP;

% n1120 --> nL1220
PUSH;
ASSERT n1120 AND set_1 AND set1_try1 AND try_1 AND init_2 ;

QUERY nL1220;
POP;

% nL1220 --> n1320
PUSH;
ASSERT nL1220 AND try_1 AND try1_enter1 AND enter_1 AND init_2 ;

QUERY n1320;
POP;

% n1320 --> n1020
PUSH;
ASSERT n1320 AND enter_1 AND enter1_init1 AND init_1 AND init_2 ;

QUERY n1020;
POP;

% n1121 --> nL1221
PUSH;
ASSERT n1121 AND set_1 AND set1_try1 AND try_1 AND set_2 ;

QUERY nL1221 ;
POP;

% nL1221 --> nR1222
PUSH;
ASSERT nL1221 AND try_1 AND set_2 AND set2_try2 AND try_2 ;

QUERY nR1222;
POP;

% nL1222 --> nL1220
PUSH;
ASSERT nL1222 AND try_1 AND try_2 AND try2_init2 AND init_2 ;

QUERY nL1220;
```

```
POP;

% nL1222 --> n1322
PUSH;
ASSERT nL1222 AND enter_1 AND enter1_init1 AND init_1 AND try_2 ;

QUERY n1322;
POP;

% n1322 --> n1320
PUSH;
ASSERT n1322 AND enter_1 AND try2_init2 AND init_2 AND try_2 ;

QUERY n1320 ;
POP;

% n1322 --> nL1022
PUSH;
ASSERT n1322 AND enter_1 AND enter1_init1 AND init_1 AND try_2 ;

QUERY nL1022;
POP;

%nL1022 --> nL1122
PUSH;
ASSERT nL1022 AND init_1 AND init1_set1 AND set_1 AND set_2;

QUERY nL1122;
POP;

% nL1022 --> n1020
PUSH;
ASSERT nL1022 AND init_1 AND try2_init2 AND init_2 AND try_2 ;

QUERY p1.k = 0 AND p2.k = 0 ;
POP;

%nL1122 --> nL1222
PUSH;
ASSERT nL1122 AND set_1 AND set1_try1 AND try_1 AND set_2;

QUERY nL1222;
POP;
```

```
%%%

% n1020 --> n1021
PUSH;
ASSERT n1020 AND init_2 AND init2_set2 AND set_2 AND init_1;

QUERY p1.k = 0 AND p2.k = 0 AND pos(p2.c) <= 1  ;
POP;

% n1021 --> n1121
PUSH;
ASSERT n1021 AND set_2 AND init_1 AND init1_set1 AND set_1 ;

QUERY p1.k = 0 AND p2.k = 0 AND pos(p2.c) <= 1 AND pos(p1.c) <= 1;
POP;

% n1021 --> nL1022
PUSH;
ASSERT n1021 AND set_2 AND set2_try2 AND try_2 AND init_1 ;

QUERY nL1022;
POP;

% nL1022 --> n1023
PUSH;
ASSERT nL1022 AND try_2 AND try2_enter2 AND enter_2 AND init_1 ;

QUERY n1023;
POP;

% n1023 --> n1020
PUSH;
ASSERT n1023 AND enter_2 AND enter2_init2 AND init_2 AND init_1 ;

QUERY n1020;
POP;

% n1121 --> nL1122
PUSH;
ASSERT n1121 AND set_2 AND set2_try2 AND try_2 AND set_1 ;

QUERY nL1122 ;
POP;
```

```
% nL1122 --> nR1222
PUSH;
ASSERT nL1122 AND try_2 AND set_1 AND set1_try1 AND try_1 ;

QUERY nR1222;
POP;

% nL1222 --> nL1022
PUSH;
ASSERT nL1222 AND try_2 AND try_1 AND try1_init1 AND init_1 ;

QUERY nL1022;
POP;

% nL1222 --> n1223
PUSH;
ASSERT nL1222 AND enter_2 AND enter2_init2 AND init_2 AND try_1 ;

QUERY n1223;
POP;

% n1223 --> n1023
PUSH;
ASSERT n1223 AND enter_2 AND try1_init1 AND init_1 AND try_1 ;

QUERY n1023 ;
POP;

% n1223 --> nL1220
PUSH;
ASSERT n1223 AND enter_2 AND enter2_init2 AND init_2 AND try_1 ;

QUERY nL1220;
POP;

%nL1220 --> nL1221
PUSH;
ASSERT nL1220 AND init_2 AND init2_set2 AND set_2 AND set_1;

QUERY nL1221;
POP;

% nL1220 --> n1020
PUSH;
```

```
ASSERT nL1220 AND init_2 AND try1_init1 AND init_1 AND try_1 ;

QUERY n1020 ;
POP;

%nL1221 --> nL1222
PUSH;
ASSERT nL1221 AND set_2 AND set2_try2 AND try_2 AND set_1;

QUERY nL1222;
POP;
```

For the N-process version of Fischer's protocol: related to the whole system

Proof obligations

For the initial condition between $\mathcal{X}_p$ and $\Delta 1_p$

$$\begin{array}{ll} & cs = 0 \wedge k = 0 \wedge (\forall i \in 1...N : p[i].loc = 0) \\ \Rightarrow & (\forall i \in 1...N : p[i].loc \in \{0,1,2\}) \wedge cs = 0 \wedge k = 0 \end{array}$$

For a set of transitions of process i to control location 1 and we call this set of transitions *try* in CVC-Lite codes reference.

$$\begin{array}{ll} & N_0 \wedge \ k' = k \wedge \ cs' = k \\ & \wedge \ p' = [p \ EXCEPT \ (p[i]'.loc = 1 \wedge p[i]'.c \leq 1)] \\ \Rightarrow & N_0' \vee N_1' \end{array}$$

For a set of transitions of process i from control location 1 to control location 2 and we call this set of transitions *set* in CVC-Lite codes reference.

$$\begin{array}{ll} & N_0 \wedge \ k' = i \wedge \ cs' = cs \\ & \wedge \ p' = [p \ EXCEPT \ (p[i]'.loc = 2 \wedge p[i]'.c \leq 2)] \\ \Rightarrow & N_0' \vee N_1' \end{array}$$

For a set of transitions of process i from control location 2 to control location 3 and we call this set of transitions *enter* in CVC-Lite codes reference.

$$\begin{array}{ll} & N_1 \wedge \ k' = k \wedge \ cs' = k \\ & \wedge \ p' = [p \ EXCEPT \ (p[i]'.loc = 3 \wedge p[i]'.c \geq 2)] \\ \Rightarrow & N_1' \vee N_2' \end{array}$$

For a transition of certain process i from control location 2 to control location 0 and we call this transition *abort* in CVC-Lite codes reference. The interesthing is that when the process takes an abort transition, the rest of other processes

j are moving into the location 2 and the shared variable k is updated by those processes IDs

$$\begin{array}{ll} & N_1 \wedge\ k' = j \wedge\ cs' = cs \\ & \wedge\ p' = [p\ EXCEPT\ (p[k]'.loc = 2 \wedge p[k]'.c \leq 2 \wedge p[i]'.loc = 0 \wedge p[i]'.c = 0)] \\ \Rightarrow & N_1' \end{array}$$

For a set of transitions of process i from control location 3 to control location 0 and we call this set of transitions *exit* in CVC-Lite codes reference.

$$\begin{array}{ll} & N_2 \wedge\ k' = 0 \wedge\ cs' = 0 \\ & \wedge\ p' = [p\ EXCEPT\ (p[i]'.loc = 0 \wedge p[i]'.c = 0)] \\ \Rightarrow & N_2' \vee N_0' \end{array}$$

CVC-Lite Codes

```
%% PDT has three nodes (n0-n1-n2) %%

Process : TYPE = ARRAY INT OF [# loc: INT, c:REAL, cs:INT #];

p, p1 : Process ;

i, k, k1, N : INT ;

%%% Nodes %%%

N0 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND (p[i].loc=0 OR p[i].loc=1 OR p[i].loc=2)
          AND p[i].cs=0 AND k=0 ;

N1_A1 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND (p[i].loc=0 OR p[i].loc=1 OR p[i].loc=2) ;

N1_A2 : BOOLEAN = EXISTS (i:INT) : 1 <= i AND i <= N
          AND p[i].loc=2 AND k = i AND p[i].cs = 0 ;

N1_A3 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND p[i].loc=1
          => (p[k].c <= p[i].c AND p[i].c <= 1) ;

N1_AA : BOOLEAN = N1_A1 AND N1_A2 AND N1_A3 ;

N2_A1 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND (p[i].loc=0 OR p[i].loc=2 OR p[i].loc=3) ;
```

```
N2_A2 : BOOLEAN = EXISTS (i:INT) : 1 <= i AND i <= N
          AND p[i].loc=3 AND k = i AND p[i].cs = k ;

N2_AA : BOOLEAN = N2_A1 AND N2_A2 ;

%%% Nodes-prime %%%

N01 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND (p1[i].loc=0 OR p1[i].loc=1 OR p1[i].loc=2)
          AND p1[i].cs=0 AND k1=0 ;

N11_A1 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND (p1[i].loc=0 OR p1[i].loc=1 OR p1[i].loc=2) ;

N11_A2 : BOOLEAN = EXISTS (i:INT) : 1 <= i AND i <= N
          AND p1[i].loc=2 AND k1 = i AND p1[i].cs = 0 ;

N11_A3 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
         AND p[i].loc=1
         => (p1[k].c <= p1[i].c AND p1[i].c <= 1);

N11_AA : BOOLEAN = N11_A1 AND N11_A2 AND N11_A3 ;

N21_A1 : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND (p1[i].loc=0 OR p1[i].loc=2 OR p1[i].loc=3) ;

N21_A2 : BOOLEAN = EXISTS (i:INT) : 1 <= i AND i <= N
          AND p1[i].loc=3 AND k1 = i AND p1[i].cs = k1 ;

N21_AA : BOOLEAN = N21_A1 AND N21_A2 ;

%%% XTGs %%%

Init : BOOLEAN = FORALL (i:INT) : 1 <= i AND i <= N
          AND p[i].loc = 0  AND p[i].cs=0 AND k=0 ;

%%% Init %%%
PUSH;
ASSERT  Init;
QUERY N0;
POP;

%%% N0 - try -> N11
PUSH;
```

```
ASSERT N0 AND Init AND k1=k AND (1 <= i AND i <= N) AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
       => (p1[j] = p[j] AND p1[i].loc=1 AND p1[i].c <=1) ;
QUERY N11_AA;
POP;

%%% N0 - set -> N11
PUSH;
ASSERT N0 AND k1=i AND (1 <= i AND i <= N) AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
        => (p1[j] = p[j] AND p1[i].loc=2 AND p1[i].c <=2) ;
QUERY N11_AA;
POP;

%%% N1 - enter -> N21
PUSH;
ASSERT N1_AA AND k1=k AND (1 <= i AND i <= N) AND
   FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
    => ( p1[j] = p[j] AND p1[i].loc=3 AND
         p1[i].c >=2 AND p1[i].cs = k1 ) ;
QUERY N21_AA;
POP;

%%% N1 - abort -> N11
PUSH;
ASSERT  N1_AA AND (1 <= i AND i <= N) AND k1 /= i AND
 EXISTS (j:INT) : (j /= i AND 1 <= j AND j <= N) AND k1=j
    => (FORALL (t:INT) : (t /= j AND 1 <= t AND t <= N)
         => (p1[t] = p[t] AND p1[k1].loc = 2 AND p1[k1].c <= 2
             AND p1[i].loc = 0 AND p1[i].c = 0)) ;
QUERY N11_AA;
POP;

%%% N2 - exit -> N01
PUSH;
ASSERT  N2_AA AND k1=0 AND (1 <= i AND i <= N) AND
    FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
       => (p1[j] = p[j] AND p1[i].loc=0 AND
           p1[i].c = 0 AND p1[i].cs = 0) ;
QUERY N01;
POP;
```

For the N-process version of Fischer's protocol: related to the universal properties

Proof obligations

For the initial condition between $\mathcal{X}_p$ and $\Delta 2_p$

$$\begin{array}{rl} & (\forall i, j \in 1...N \wedge (i \neq j) : p[i].loc = 0 \wedge p[j].loc = 0) \\ & \wedge\ cs = 0 \wedge k = 0 \\ \Rightarrow & (\exists i \in 1...N : p[i].loc \in \{0,1,2\})\ \wedge cs = 0 \wedge k = 0 \\ & \wedge\ (\forall j \in 1...N \wedge (i \neq j) : p[j].loc \in \{0,1,2\}) \end{array}$$

For an edge from N_0 to N_{R1}: for a set of transitions set_i in $\mathcal{X}_p$

$$\begin{array}{rl} & N_0 \wedge\ k' = i \wedge\ cs' = cs \\ & \wedge\ p' = [p\ EXCEPT\ (p[i]'.loc = 2 \wedge p[i]'.c \leq 2 \wedge p[j]'.loc = 1 \wedge p[j]'.c \leq 1)] \\ \Rightarrow & N'_0 \vee N'_{R1} \end{array}$$

For an edge from N_{R1} to N_{R2}: for a set of τ-transitions of process i after set_i in $\mathcal{X}_p$

$$\begin{array}{rl} & N_{R1} \wedge\ k' = k \wedge\ cs' = cs \\ & \wedge\ p' = [p\ EXCEPT\ p[j]'.loc \in \{0,2\}] \\ \Rightarrow & N'_{R1} \vee N'_{R2} \end{array}$$

For an edge from N_{R2} to N_{R3}: for a set of transitions $enter_i$ in $\mathcal{X}_p$

$$\begin{array}{rl} & N_{R2} \wedge\ k' = k \wedge\ cs' = k \\ & \wedge\ p' = [p\ EXCEPT\ (p[i]'.loc = 3 \wedge p[i]'.c \geq 2)] \\ \Rightarrow & N'_{R2} \vee N'_{R3} \end{array}$$

For an edge from N_{R1} to N_{L1}: for a set of transitions set_j and $abort_i$ in $\mathcal{X}_p$

$$\begin{array}{rl} & N_{R1} \wedge\ k' = j \wedge\ cs' = cs \\ & \wedge\ p' = [p\ EXCEPT\ (p[k]'.loc = 2 \wedge p[k]'.c \leq 2 \wedge p[i]'.loc = 0 \wedge p[i]'.c = 0)] \\ \Rightarrow & N'_{R1} \vee N'_{L2} \end{array}$$

For an edge from N_{R3} to N_0: for a set of transitions $exit_i$ in $\mathcal{X}_p$

$$\begin{array}{rl} & N_{R3} \wedge\ k' = 0 \wedge\ cs' = 0 \\ & \wedge\ p' = [p\ EXCEPT\ (p[i]'.loc = 0 \wedge p[i]'.c = 0)] \\ \Rightarrow & N'_{R3} \vee N'_0 \end{array}$$

For an edge from N_0 to N_{L1}: for a set of transitions set_j in $\mathcal{X}_p$

$$\begin{array}{rl} & N_0 \wedge\ k' = j \wedge\ cs' = cs \\ & \wedge\ p' = [p\ EXCEPT\ (p[k]'.loc = 2 \wedge p[k]'.c \leq 2 \wedge p[i]'.loc = 1 \wedge p[i]'c \leq 1)] \\ \Rightarrow & N'_0 \vee N'_{L1} \end{array}$$

For an edge from N_{L1} to N_{L2}: for a set of transitions $enter_i$ in $\mathcal{X}_p$

$$\begin{array}{rl} & N_{L1} \wedge\ k' = k \wedge\ cs' = k \\ & \wedge\ p' = [p\ EXCEPT\ (p[k]'.loc = 3 \wedge p[k]'.c \geq 2)] \\ \Rightarrow & N'_{L1} \vee N'_{L2} \end{array}$$

For an edge from N_{L1} to N_{R2}: for a set of transitions set_i in $\mathcal{X}_p$

$$\begin{array}{l} N_{R1} \wedge\ k' = i \wedge\ cs' = cs \\ \wedge\ p' = [p\ EXCEPT\ (p[i]'.loc = 2 \wedge p[i]'.c \leq 2 \wedge p[j]'.loc = 0 \wedge p[j]'.c = 0)] \\ \Rightarrow\ N'_{L1} \vee N'_{R2} \end{array}$$

For an edge from N_{L2} to N_0: for a set of transitions $exit_j$ in $\mathcal{X}_p$

$$\begin{array}{rl} & N_{L2} \wedge\ k' = 0 \wedge\ cs' = 0 \\ & \wedge\ p' = [p\ EXCEPT\ (p[j]'.loc = 0 \wedge p[j]'.c = 0)] \\ \Rightarrow & N'_{L2} \vee N'_0 \end{array}$$

CVC-Lite Codes

```
Process : TYPE = ARRAY INT OF [# loc: INT, c:REAL, cs:INT #];

p, p1 : Process ;

i, k, k1, N : INT ;

%%% Righthand Nodes %%%

N0_A1 : BOOLEAN = (k = 0) AND (1 <= i AND i <= N)
     AND (p[i].loc=0 OR p[i].loc=1 OR p[i].loc=2)
     AND p[i].cs=0 ;

N0_A2 : BOOLEAN = (1 <= i AND i <= N) AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
        AND (p[j].loc=0 OR p[j].loc=1 OR p[j].loc=2)
        AND p[j].cs=0;

N0_AA : BOOLEAN = N0_A1 AND N0_A2 ;

NR1_A1 : BOOLEAN = (k = i) AND (1 <= i AND i <= N)
           AND p[i].loc=2 AND p[i].cs = 0 ;

NR1_A2 : BOOLEAN = (1 <= i AND i <= N) AND
       FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
          AND (p[j].loc=0 OR p[j].loc=1 OR p[j].loc=2)
```

```
          AND p[j].cs=0;

NR1_A3 : BOOLEAN = (1 <= i AND i <= N) AND
 FORALL (j:INT):(j /= i AND 1 <= j AND j <= N AND p[j].loc=1)
       => (p[i].c <= p[j].c AND p[j].c <= 1);

NR1_AA : BOOLEAN = NR1_A1 AND NR1_A2 AND NR1_A3 ;

NR2_A1 : BOOLEAN = (1 <= i AND i <= N) AND k=i
       AND p[i].loc=3 AND p[i].cs=0 AND p[i].c <=2 ;

NR2_A2 : BOOLEAN = (1 <= i AND i <= N) AND
    FORALL (j:INT) : j /= i AND 1 <= j AND j <= N
       AND (p[j].loc=0 OR p[j].loc=2) AND p[j].cs=0 ;

NR2_AA : BOOLEAN = NR2_A1 AND NR2_A2 ;

NR3_AA : BOOLEAN = (1 <= i AND i <= N) AND k=i AND
    (p[i].loc = 3 AND p[i].c >= 2 AND p[i].cs=k) AND
  FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N) AND
    (p[j].loc=0 OR p[j].loc=2) AND p[j].cs = k;


%%% Righthand Nodes-prime %%%
N01_A1 : BOOLEAN = (k1 = 0) AND (1 <= i AND i <= N)
     AND (p1[i].loc=0 OR p1[i].loc=1 OR p1[i].loc=2)
     AND p1[i].cs=0 ;

N01_A2 : BOOLEAN = (1 <= i AND i <= N) AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
        AND (p1[j].loc=0 OR p1[j].loc=1 OR p1[j].loc=2)
        AND p1[j].cs=0;

N01_AA : BOOLEAN = N01_A1 AND N01_A2 ;

NR11_A1 : BOOLEAN = (k1 = i) AND (1 <= i AND i <= N)
           AND p1[i].loc=2 AND p1[i].cs = 0 ;

NR11_A2 : BOOLEAN = (1 <= i AND i <= N) AND
       FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
          AND (p1[j].loc=0 OR p1[j].loc=1 OR p1[j].loc=2)
          AND p1[j].cs=0;

NR11_A3 : BOOLEAN = (1 <= i AND i <= N) AND
```

```
 FORALL (j:INT):(j /= i AND 1 <= j AND j <= N AND p1[j].loc=1)
       => (p1[i].c <= p1[j].c AND p1[j].c <= 1);

NR11_AA : BOOLEAN = NR11_A1 AND NR11_A2 AND NR11_A3 ;

NR21_A1 : BOOLEAN = (1 <= i AND i <= N) AND k1=i
       AND p1[i].loc=3 AND p1[i].cs=0 AND p1[i].c <=2 ;

NR21_A2 : BOOLEAN = (1 <= i AND i <= N) AND
    FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
       AND (p1[j].loc=0 OR p1[j].loc=2) AND p1[j].cs=0 ;

NR21_AA : BOOLEAN = NR21_A1 AND NR21_A2 ;

NR31_AA : BOOLEAN = (1 <= i AND i <= N) AND k1=i AND
    (p1[i].loc = 3 AND p1[i].c >= 2 AND p1[i].cs=k1) AND
  FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N) AND
    (p1[j].loc=0 OR p1[j].loc=2) AND p1[j].cs = k1;

%%% Lefthand Nodes %%%
NL1_A1 : BOOLEAN = (1 <= i AND i <= N) AND p[i].cs=0
        AND (p[i].loc=0 OR p[i].loc=1 OR p[i].loc=2);

NL1_A2 : BOOLEAN = (1 <= i AND i <= N) AND
    FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
     AND (p[j].loc=0 OR p[j].loc=1 OR p[j].loc=2)
     AND p[j].cs=0;

NL1_A3 : BOOLEAN = (1 <= i AND i <= N) AND
 FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N AND p[j].loc=1)
   => (p[k].c <= p[j].c AND p[j].c <= 1);

NL1_A4 : BOOLEAN = (1 <= i AND i <= N) AND
    FORALL (j:INT) : j /= i AND 1 <= j AND j <= N
     AND k=j AND p[k].loc = 2 AND p[k].c <= 2;

NL1_AA : BOOLEAN = NL1_A1 AND NL1_A2 AND NL1_A3 AND NL1_A4  ;


NL2_AA : BOOLEAN = (1 <= i AND i <= N)
      AND (p[i].loc=0 OR p[i].loc=2) AND
  FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
      AND (p[j].loc=0 OR p[j].loc=2 OR p[j].loc=3)
      AND k=j AND p[j].cs = k AND p[i].cs = k ;
```

```
%%% Lefthand Nodes-prime %%%
NL11_A1 : BOOLEAN = (1 <= i AND i <= N) AND p1[i].cs=0
        AND (p1[i].loc=0 OR p1[i].loc=1 OR p1[i].loc=2);

NL11_A2 : BOOLEAN = (1 <= i AND i <= N) AND
    FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
     AND (p1[j].loc=0 OR p1[j].loc=1 OR p1[j].loc=2)
     AND p1[j].cs=0;

NL11_A3 : BOOLEAN = (1 <= i AND i <= N) AND
 FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N AND p1[j].loc=1)
   => (p1[k].c <= p1[j].c AND p1[j].c <= 1);

NL11_A4 : BOOLEAN = (1 <= i AND i <= N) AND
    FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
     AND k1=j AND p1[k1].loc = 2 AND p1[k1].c <= 2;

NL11_AA : BOOLEAN = NL11_A1 AND NL11_A2 AND NL11_A3 AND NL11_A4  ;


NL21_AA : BOOLEAN = (1 <= i AND i <= N)
      AND (p1[i].loc=0 OR p1[i].loc=2) AND
  FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
      AND (p1[j].loc=0 OR p1[j].loc=2 OR p1[j].loc=3)
      AND k1=j AND p1[j].cs = k1 AND p1[i].cs = k1 ;

%%% XTGs %%%

Init : BOOLEAN = (k = 0) AND (1 <= i AND i <= N )
    AND p[i].loc=0 AND p[i].cs = 0 AND p[i].c=0 AND
   FORALL (j:INT) : j /= i AND (1 <= j AND j <= N)
       AND p[j].loc = 0 AND p[j].cs = 0 AND p[j].c=0;

%%% Init %%%
PUSH;
ASSERT  Init;
QUERY N0_AA;
POP;

%%% N0 - seti -> NR1
PUSH;
ASSERT N0_AA AND k1=i AND (1 <= i AND i <= N) AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
```

```
      => (p1[j] = p[j] AND p1[i].loc=2 AND p1[i].c <=2) ;

QUERY NR11_AA;
POP;

%%%%% NR1 - tau -> NR2
PUSH;
ASSERT NR1_AA AND k1=k AND (1 <= i AND i <= N) AND
  FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
   => (p1[j] = p[j] AND p1[i] = p[i]) ;

QUERY NR21_AA;
POP;

%%% NR2 - enteri -> NR3
PUSH;
ASSERT NR2_AA AND k1=k AND (1 <= i AND i <= N) AND
 FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
   => ( p1[j] = p[j] AND p1[i].loc=3 AND
        p1[i].c >=2 AND p1[i].cs=k1 AND p1[j].cs=k1) ;

QUERY NR31_AA;
POP;

%%% NR3- exiti -> N0
PUSH;
ASSERT NR3_AA AND k1=0 AND (1 <= i AND i <= N) AND
 FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
  => ( p1[j] = p[j] AND p1[i].loc=0 AND
       p1[i].c=0 AND p1[i].cs=0 AND p1[j].cs=0) ;

QUERY N01_AA;
POP;

%% NR1 - setj/aborti -> NL1
PUSH;
ASSERT NR1_AA AND (1 <= i AND i <= N) AND k1 /= i AND
  EXISTS (j:INT) : (j /= i AND 1 <= j AND j <= N) AND k1=j
    => ( FORALL (t:INT) : (t /= j AND 1 <= t AND t <= N)
      => ( p1[t] = p[t] AND p1[k1].loc=2 AND p1[k1].c <= 2 AND
           p1[i].loc = 0 AND p1[i].c = 0 )) ;

QUERY NL11_AA;
POP;
```

```
%%% N0 - setj -> NL1 %%%
PUSH;
ASSERT N0_AA AND (1 <= i AND i <= N) AND
  EXISTS (j:INT) : (j /= i AND 1 <= j AND j <= N AND k1=j)
    => ( FORALL (t:INT) : (t /= j AND 1 <= t AND t <= N)
      => ( p1[t] = p[t] AND p1[k1].loc=2 AND p1[k1].c <=2 AND
           p1[i].loc = 0 AND p1[i].c <= 1) ) ;

QUERY NR11_AA;
POP;

%%% NL1 - enterj -> NL2
PUSH;
ASSERT NL1_AA AND k1=k AND (1 <= i AND i <= N) AND
 EXISTS (j:INT) : (j /= i AND 1 <= j AND j <= N)
   => (FORALL (t:INT) : (t /= j AND 1 <= t AND t <= N)
     => (p1[t] = p[t] AND p1[k1].loc=3 AND
         p1[k1].c >=2 AND p1[i].cs=k1 AND p1[j].cs=k1)) ;

QUERY NL21_AA;
POP;

%%% NL2- exiti -> N0
PUSH;
ASSERT NL2_AA AND k1=0 AND (1 <= i AND i <= N) AND
  EXISTS (j:INT) : (j /= i AND 1 <= j AND j <= N)
    => (FORALL (t:INT) : (t /= j AND 1 <= t AND t <= N)
      => (p1[t] = p[t] AND p1[j].loc=0 AND
          p1[j].c = 0 AND p1[i].cs=0 AND p1[j].cs=0)) ;

QUERY N01_AA;
POP;

%% NL1 - seti -> NR2/NR1
PUSH;
ASSERT NL1_AA AND (1 <= i AND i <= N) AND k1=i AND
  EXISTS (j:INT) : (j /= i AND 1 <= j AND j <= N AND k1 /= j)
    => (FORALL (t:INT) : (t /= j AND 1 <= t AND t <= N)
      => (p1[t] = p[t] AND p1[k1].loc=2 AND p1[k1].c <= 2)) ;

QUERY NR21_AA;
POP;
```

```
%% Property Verification  %%
PUSH;
ASSERT N0_AA ;
QUERY p[i].cs = 0 OR (p[i].cs = i) ;
POP;

PUSH;
ASSERT (NR1_AA OR NR2_AA OR NR3_AA) ;
QUERY p[i].cs = 0 OR (p[i].cs = i) ;
POP;

PUSH;
ASSERT (NL1_AA OR NL2_AA) ;
QUERY EXISTS (j:INT) : (1 <= j AND j <= N )
      AND p[j].cs = 0 OR (p[j].cs = i) ;
POP;
```

Appendix B

CVC-Lite code of Railway Crossing System

This appendix contains the comeplete proof of conformance between XTGs specification of railway crossing system and PDTs by reproducing CVC-Lite input.

For the 2-process version

```
process_train: TYPE = [#id:INT, loc:INT, c:REAL#];

process_controller: TYPE = [#loc:INT, cnt:INT#];

process_gate: TYPE = [#loc:INT, c:REAL#];

approach_O, approach_I : BOOLEAN;
close_O, close_I : BOOLEAN ;
open_O, open_I : BOOLEAN ;
leave_O, leave_I : BOOLEAN ;

t1, t2: process_train;

ctl: process_controller;

g: process_gate;

pos: REAL -> REAL = LAMBDA (x:REAL): IF x>=0 THEN x
                                     ELSE (-x)   ENDIF;
```

```
% possible locations for train processes

far_1 : BOOLEAN =
         t1.id = 1 AND t1.loc = 0 AND pos(t1.c) < 20;

near_1 : BOOLEAN =
         t1.id = 1 AND t1.loc = 1 AND pos(t1.c) < 6 ;

on_1 : BOOLEAN =
         t1.id = 1 AND t1.loc = 2 AND pos(t1.c) < 4 ;

far_2 : BOOLEAN =
         t2.id = 2 AND t2.loc = 0 AND pos(t2.c) < 20;

near_2 : BOOLEAN =
         t2.id = 2 AND t2.loc = 1 AND pos(t2.c) < 6 ;

on_2 : BOOLEAN =
         t2.id = 2 AND t2.loc = 2 AND pos(t2.c) < 4 ;

% possible locations for controller

free : BOOLEAN = ctl.loc = 0 AND ctl.cnt = 0 ;

busy : BOOLEAN = ctl.loc  = 1 AND ctl.cnt >= 1 ;

% possible locations for gate

open : BOOLEAN = g.loc = 0 AND pos(g.c) >= 0 ;

closing : BOOLEAN = g.loc = 1  AND pos(g.c) < 1 ;

close : BOOLEAN = g.loc = 2 AND pos(g.c) >= 1 ;

opening :BOOLEAN = g.loc = 3 AND pos(g.c) < 1 ;

% possible transitions for tains

far1_near1 : BOOLEAN =  pos(t1.c) >= 5 AND approach_O ;

near1_on1 : BOOLEAN =  pos(t1.c) >= 6 ;

on1_far1 : BOOLEAN =  pos(t1.c) >= 4 AND leave_O ;
```

```
far2_near2 : BOOLEAN =  pos(t2.c) >= 5 AND approach_O ;

near2_on2 : BOOLEAN =  pos(t2.c) >= 6 ;

on2_far2 : BOOLEAN =  pos(t2.c) >= 4 AND leave_O ;

% possible transitions for controller

free_busy : BOOLEAN =  ctl.cnt >= 1 AND close_O ;

busy_free : BOOLEAN =  ctl.cnt < 1 AND open_O ;

% possible transitions for gate

open_closing : BOOLEAN = close_I ;

closing_close : BOOLEAN = pos(g.c) >= 1 ;

close_opening : BOOLEAN = open_I ;

opening_open : BOOLEAN = pos(g.c) >= 1 ;

%%% predicates of set of nodes %%%

n_init : BOOLEAN =  t1.loc = 0 AND t2.loc = 0 AND free AND open
         AND pos(t1.c) < 20 AND pos(t2.c) < 20 AND ctl.cnt = 0;

%%% lefthand sides %%%

n_1 : BOOLEAN = t1.loc = 1 AND t2.loc = 0 AND free AND open
        AND pos(t1.c) < 20 AND pos(t2.c) < 20 AND ctl.cnt=0 ;

n_14 : BOOLEAN = t1.loc = 1 AND t2.loc = 0 AND busy AND closing
        AND pos(t1.c) <= pos(t2.c)
        AND ctl.cnt = 1 AND pos(g.c) <= 1 ;

n_31 : BOOLEAN = t1.loc = 1 AND t2.loc = 1 AND busy AND closing
        AND pos(t1.c) >=  pos(t2.c)
        AND ctl.cnt = 2 AND pos(g.c) <= 1 ;

n_4 : BOOLEAN = t1.loc = 2 AND t2.loc = 0 AND busy AND close
        AND pos(t1.c) <=  pos(t2.c)
        AND ctl.cnt = 1 AND pos(g.c) >= 1 ;
```

```
n_6 : BOOLEAN = t1.loc = 2 AND t2.loc = 1 AND busy AND close
      AND pos(t1.c) >=  pos(t2.c)
      AND ctl.cnt = 2 AND pos(g.c) >= 1 ;

n_61 : BOOLEAN = t1.loc = 0 AND t2.loc = 1 AND busy AND close
      AND pos(t1.c) <=  pos(t2.c)
      AND ctl.cnt = 1 AND pos(g.c) >= 1 ;

n_36 : BOOLEAN = t1.loc = 1 AND t2.loc = 1 AND busy AND close
      AND pos(t1.c) >=  pos(t2.c)
      AND ctl.cnt = 2 AND pos(g.c) >= 1 ;

n_81 : BOOLEAN = t1.loc = 2 AND t2.loc = 1  AND busy AND close
      AND pos(t1.c) >=  pos(t2.c)
      AND ctl.cnt = 2 AND pos(g.c) >= 1 ;

n_41 : BOOLEAN = t1.loc = 0 AND t2.loc = 0 AND free AND opening
      AND pos(t1.c) <=  pos(t2.c)
      AND ctl.cnt = 0 AND pos(g.c) <= 1 ;

n_42 : BOOLEAN = t1.loc = 0 AND t2.loc = 1 AND free AND opening
      AND pos(t1.c) >=  pos(t2.c)
      AND ctl.cnt = 1 AND pos(g.c) <= 1 ;

%%% righthand side %%%
n_2 : BOOLEAN = t1.loc = 0 AND t2.loc = 1 AND free AND open
      AND pos(t1.c) >= pos(t2.c) AND ctl.cnt = 1 ;

n_32 : BOOLEAN = t1.loc = 1 AND t2.loc = 1 AND busy AND closing
      AND pos(t1.c) <= pos(t2.c) AND ctl.cnt = 2
      AND pos(g.c) <= 1 ;

n_25 : BOOLEAN = t1.loc = 0 AND t2.loc = 1 AND busy AND closing
      AND pos(t1.c) >= pos(t2.c) AND ctl.cnt = 1
      AND pos(g.c) <= 1 ;

n_37 : BOOLEAN = t1.loc = 1 AND t2.loc = 1 AND busy AND close
      AND pos(t1.c) <=  pos(t2.c) AND ctl.cnt = 2
      AND pos(g.c) >= 1 ;

n_5 : BOOLEAN = t1.loc = 0 AND t2.loc = 2 AND busy AND close
      AND pos(t1.c) >= pos(t2.c) AND ctl.cnt = 1
      AND pos(g.c) >= 1 ;
```

```
n_7 : BOOLEAN = t1.loc = 1 AND t2.loc = 2 AND busy AND close
      AND pos(t1.c) <= pos(t2.c) AND ctl.cnt = 2
      AND pos(g.c) >= 1 ;

n_82 : BOOLEAN = t1.loc = 2 AND t2.loc = 2 AND busy AND close
      AND pos(t1.c) <= pos(t2.c) AND ctl.cnt = 2
      AND pos(g.c) >= 1 ;

n_71 : BOOLEAN = t1.loc = 1 AND t2.loc = 0 AND busy AND close
      AND pos(t1.c) <= pos(t2.c) AND ctl.cnt = 1
      AND pos(g.c) >= 1 ;

n_51 : BOOLEAN = t1.loc = 0 AND t2.loc = 0 AND free AND opening
      AND pos(t1.c) >= pos(t2.c) AND ctl.cnt = 0
      AND pos(g.c) <= 1 ;

n_52 : BOOLEAN = t1.loc = 1 AND t2.loc = 0 AND free AND opening
      AND pos(t1.c) <= pos(t2.c) AND ctl.cnt = 1
      AND pos(g.c) <= 1 ;

%%% initial
PUSH;
ASSERT far_1 AND far_2 AND free AND open ;
QUERY n_init;
POP;

%%% consider nodes in left-hand side.
% n_init -> n_1 : (far1 far2) -> (near1 far2)
PUSH;
ASSERT n_init AND far_1 AND far1_near1 AND near_1 AND far_2
       AND free AND open  ;
QUERY n_init OR n_1;
POP;

% n_1 -> n31 : (near1 far2) -> (near1 near2)
PUSH;
ASSERT n_1 AND near_1 AND far2_near2 AND near_2 AND free_busy AND busy
       AND open_closing AND closing ;
QUERY  n_31;
POP;

% n_1 -> n_14 : (near1 far2) -> (near1 far2)
PUSH;
ASSERT n_1 AND near_1 AND far_2 AND free AND free_busy AND busy
```

```
       AND open_closing AND closing ;
QUERY  n_14;
POP;

% n_14 -> n_4 : (near1 far2) -> (on1 far2)
PUSH;
ASSERT n_14 AND near_1 AND near1_on1 AND on_1 AND far_2 AND busy
       AND closing_close AND close;
QUERY  n_4;
POP;

% n_14 -> n_31 : (near1 far2) -> (near1 near2)
PUSH;
ASSERT n_14 AND far_2 AND far2_near2 AND near_2 AND near_1 AND busy
       AND closing ;
QUERY  n_31;
POP;

% n_31 -> n_36 : (near1 near2) -> (near1 near2)
PUSH;
ASSERT n_31 AND near_1 AND near_2 AND busy
     AND closing_close AND close ;
QUERY n_36;
POP;

% n_36 -> n_6 : (near1 near2) -> (on1 near2)
PUSH;
ASSERT n_36 AND near_1 AND near1_on1 AND on_1 AND near_2
      AND busy AND close ;
QUERY n_6;
POP;

% n_4 -> n_6 : (on1 far2) -> (on1 near2)
PUSH;
ASSERT n_4 AND far_2 AND far2_near2 AND near_2 AND on_1
     AND busy AND close ;
QUERY n_6;
POP;

% n_6 -> n_61 : (on1 near2) -> (far1 near2)
PUSH;
ASSERT n_6 AND on_1 AND on1_far1 AND far_1 AND near_2
      AND busy AND close ;
QUERY  n_61;
```

```
POP;

% n_6 -> n_81 : (on1 near2) -> (on1 on2)
PUSH;
ASSERT n_6 AND near2_on2 AND on_2 AND on_1
      AND busy AND close ;
QUERY n_81;
POP;

% n_61 -> n_37 : (far1 near2) -> (near1 near2)
PUSH;
ASSERT n_61 AND far_1 AND far1_near1 AND near_1 AND near_2
       AND busy AND close ;
QUERY  n_37;
POP;

% n_61 -> n_5 : (far1 near2) -> (far1 on2)
PUSH;
ASSERT n_61 AND near_2 AND near2_on2 AND on_2 AND far_1
       AND busy AND close ;
QUERY  n_5;
POP;

% n_81 -> n_5 : (on1 on2) -> (far1 on2)
PUSH;
ASSERT n_81 AND on_1 AND on1_far1 AND far_1 AND on_2
       AND busy AND close ;
QUERY  n_5;
POP;

% n_4 -> n_41 : (on1 far2) -> (far1 far2)
PUSH;
ASSERT n_4 AND on_1 AND on1_far1 AND far_1 AND far_2
      AND busy AND busy_free AND free
      AND close AND close_opening AND opening ;
QUERY  n_41;
POP;

% n_41 -> n_1 : (far1 far2) -> (near1 far2)
PUSH;
ASSERT n_41 AND far_1 AND far1_near1 AND near_1 AND far_2
      AND opening AND opening_open AND open AND free ;
QUERY  n_1;
POP;
```

```
% n_41 -> n_42 : (far1 far2) -> (far1 near2)
PUSH;
ASSERT n_41 AND far_2 AND far2_near2 AND near_2 AND far_1
       AND free AND opening ;
QUERY n_42;
POP;

% n_42 -> n_2 : (far1 near2) -> (far1 near2)
PUSH;
ASSERT n_42 AND far_1 AND near_2 AND free
       AND opening AND opening_open AND open ;
QUERY n_2;
POP;

% n_41 -> n_init : (far1 far2) -> (far1 far2)
PUSH;
ASSERT n_41 AND far_1 AND far_2 AND free
       AND opening_open AND open ;
QUERY  n_init;
POP;

%%% consider nodes in righthand side.
% n_init -> n_2 : (far1 far2) -> (far1 near2)
PUSH;
ASSERT n_init AND far_1 AND far2_near2 AND near_2
       AND free AND open  ;
QUERY n_init OR n_2;
POP;

% n_2 -> n32 : (far1 near2) -> (near1 near2)
PUSH;
ASSERT n_2 AND far_1 AND far1_near1 AND near_1
       AND free AND free_busy AND busy
       AND open_closing AND closing ;
QUERY  n_32;
POP;

% n_2 -> n_25 : (far1 near2) -> (far1 near2)
PUSH;
ASSERT n_2 AND far_1 AND near_2
       AND free AND free_busy AND busy
       AND open_closing AND closing ;
QUERY  n_25;
```

```
POP;

% n_25 -> n_5 : (far1 near2) -> (far1 on2)
PUSH;
ASSERT n_25 AND near_2 AND near2_on2 AND on_2 AND far_1
       AND busy AND closing_close AND close ;
QUERY  n_5;
POP;

% n_25 -> n_32 : (far1 near2) -> (near1 near2)
PUSH;
ASSERT n_25 AND far_1 AND far1_near1 AND near_1 AND near_2
       AND busy AND closing ;
QUERY  n_32;
POP;

% n_32 -> n_37 : (near1 near2) -> (near1 near2)
PUSH;
ASSERT n_32 AND near_1 AND near_2 AND busy
     AND closing AND closing_close AND close ;
QUERY n_37;
POP;

% n_37 -> n_7 : (near1 near2) -> (near1 on2)
PUSH;
ASSERT n_37 AND near_2 AND near2_on2 AND on_2 AND near_1
      AND busy AND close ;
QUERY n_7;
POP;

% n_5 -> n_7 : (far1 on2) -> (near1 on2)
PUSH;
ASSERT n_5 AND far_1 AND far1_near1 AND near_1 AND on_2
       AND busy AND close ;
QUERY n_7;
POP;

% n_7 -> n_71 : (near1 on2) -> (near1 far2)
PUSH;
ASSERT n_7 AND on_2 AND on2_far2 AND far_2 AND near_1
       AND busy AND close ;
QUERY  n_71;
POP;
```

```
% n_7 -> n_82 : (near1 on2) -> (on1 on2)
PUSH;
ASSERT n_7 AND near_1 AND near1_on1 AND on_1 AND on_2
       AND busy AND close ;
QUERY n_82;
POP;

% n_71 -> n_36 : (near1 far2) -> (near1 near2)
PUSH;
ASSERT n_71 AND far_2 AND far2_near2 AND near_2 AND near_1
       AND busy AND close ;
QUERY  n_36;
POP;

% n_71 -> n_4 : (near1 far2) -> (on1 far2)
PUSH;
ASSERT n_71 AND near_1 AND near1_on1 AND on_1 AND far_2
       AND busy AND close ;
QUERY  n_4;
POP;

% n_82 -> n_4 : (on1 on2) -> (on1 far2)
PUSH;
ASSERT n_82 AND on_2 AND on2_far2 AND far_2 AND on_1
       AND busy AND close ;
QUERY  n_4;
POP;

% n_5 -> n_51 : (far1 on2) -> (far1 far2)
PUSH;
ASSERT n_5 AND on_2 AND on2_far2 AND far_2 AND far_1
      AND busy AND busy_free AND free
      AND close AND close_opening AND opening ;
QUERY  n_51;
POP;

% n_51 -> n_2 : (far1 far2) -> (far1 near2)
PUSH;
ASSERT n_51 AND far_2 AND far2_near2 AND near_2 AND far_1
      AND opening AND opening_open AND open AND free ;
QUERY  n_2;
POP;

% n_51 -> n_52 : (far1 far2) -> (near1 far2)
```

```
PUSH;
ASSERT n_51 AND far_1 AND far1_near1 AND near_1 AND far_2
       AND free AND opening ;
QUERY  n_52;
POP;

% n_52 -> n_1 : (near1 far2) -> (near1 far2)
PUSH;
ASSERT n_52 AND far_2 AND near_1 AND free
       AND opening AND opening_open AND open ;
QUERY  n_1;
POP;

% n_51 -> n_init : (far1 far2) -> (far1 far2)
PUSH;
ASSERT n_51 AND far_1 AND far_2 AND free
     AND opening AND opening_open AND open ;
QUERY  n_init;
POP;
```

For the N-process version

Proof obligations

For the initial condition between $\mathcal{X}_p$ and $\Delta 1_p$

$$\begin{aligned}
&\forall i, j \in 1...N \wedge (i \neq j) : t[i].loc = far \wedge t[j].loc = far \wedge \\
&\wedge \;\; t[i].c < 20 \wedge t[j].c < 20 \wedge ctl.cnt = 0 \wedge g.loc = open \wedge ctl.loc = free \\
\Rightarrow \;\; &(\exists i \in 1, \ldots, N : t[i].loc = far \wedge t[i].c < 20) \\
&\wedge (\forall j \in 1, \ldots, N \wedge i \neq j : t[j].loc = far) \\
&\wedge \;\; g.loc = open \wedge ctl.loc = free \wedge ctl.cnt = 0
\end{aligned}$$

For the edge from n_0 to n_{R1} that represents the set of transitions of train-j group from control location *far* to *near*

$$\begin{aligned}
&n_0 \wedge t' = [t \text{ EXCEPT } t[j]'.loc = near \wedge t[j]'.c \leq 6] \\
&\wedge \;\; g'.loc = closing \wedge g'.c = 0 \\
&\wedge \;\; ctl'.loc = ctl.loc \wedge ctl'.cnt \leq (N-1) \\
\Rightarrow \;\; &n'_0 \vee n'_{R1}
\end{aligned}$$

For the edge from n_{R1} to n_{R2} that represents the set of transitions of train-j group from control location *near* to *on*

$$\begin{aligned}
&n_{R1} \wedge ctl'.loc = busy \wedge ctl'.cnt \leq N \\
&t' = [t \text{ EXCEPT } t[j]'.loc = on \wedge (t'[i].loc \in \{near, on\} \wedge t'[i].c \leq t'[j].c] \\
&\wedge \;\; ((g'.loc = closing \wedge g'.c < 1) \vee (g'.loc = close \wedge g'.c \geq 1)) \\
\Rightarrow \;\; &n'_{R1} \vee n'_{R2}
\end{aligned}$$

For the edge from n_{R2} to n_{R3} that represents the set of transitions of train-j group from control location *on* to *far*

$$\begin{aligned} & n_{R2} \wedge t' = [t \text{ EXCEPT } t[j]'.loc = far \wedge t[j]'.c < 20] \\ & \wedge \;\; g'.loc = close \wedge g'.c \geq 1 \\ & \wedge \;\; ctl'.loc = ctl.loc \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow \quad & n'_{R2} \vee n'_{R3} \end{aligned}$$

For the edge from n_{R3} to n_{R4} that represents the transition of train-i from control location *on* to *far*

$$\begin{aligned} & n_{R3} \wedge t' = [t \text{ EXCEPT } t[i]'.loc = far \wedge t[i]'.c \leq t[j]'.c] \\ & \wedge \;\; g'.loc = opening \wedge g'.c \leq 1 \\ & \wedge \;\; ctl'.loc = free \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow \quad & n'_{R3} \vee n'_{R4} \end{aligned}$$

For the edge from n_{R4} to n_{L1} that represents the transition of train-i from control location *far* to *near* and the rest of other trains group $t[j]$ does not move and stays in the same location *far*

$$\begin{aligned} & n_{R4} \wedge t' = [t \text{ EXCEPT } t[i]'.loc = near \wedge t[i]'.c < 6] \\ & \wedge \;\; g'.loc = open \wedge g'.c \leq 1 \\ & \wedge \;\; ctl'.loc = free \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow \quad & n'_{R4} \vee n'_{L1} \end{aligned}$$

For the edge from n_{R4} to n_0 that represents the transition of train-i at control location *far*

$$\begin{aligned} & n_{R4} \wedge \forall i, j \in 1, \ldots, N \wedge i \neq j : t'[i] = far \wedge t'[i].c < 20 \\ & \wedge \;\; t'[j].loc = far \wedge t'[j].c < 20 \\ & \wedge \;\; g'.loc = open \wedge g'.c \leq 1 \\ & \wedge \;\; ctl'.loc = free \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow \quad & n'_{R4} \vee n'_0 \end{aligned}$$

For the edge from n_{R2} to n_{L3} that represents the transition of train-i from control location *on* to *far* and the rest of other trains group $t[j]$ are in the same locations *near* or *on*

$$\begin{aligned} & n_{R2} \wedge t' = [t \text{ EXCEPT } t[i]'.loc = far \wedge t[i]'.c < 6] \\ & \wedge \;\; g'.loc = close \wedge g'.c \geq 1 \\ & \wedge \;\; ctl'.loc = ctl.loc \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow \quad & n'_{R2} \vee n'_{L3} \end{aligned}$$

For the edge from n_0 to n_{L1} that represents the transition of train-i from control location *far* to *near*

$$\begin{aligned} & n_0 \wedge t' = [t \text{ EXCEPT } t[i]'.loc = near \wedge t[i]'.c \leq 6] \\ & \wedge \;\; g'.loc = closing \wedge g'.c = 0 \\ & \wedge \;\; ctl'.loc = ctl.loc \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow \quad & n'_0 \vee n'_{L1} \end{aligned}$$

For the edge from n_{L1} to n_{L2} that represents the transition of train-i from control location *near* to *on*

$$\begin{array}{ll} & n_{L1} \wedge ctl'.loc = busy \wedge ctl'.cnt \leq N \\ & t' = [t \text{ EXCEPT } t[i]'.loc = on \wedge (t'[j].loc \in \{near, on\} \wedge t'[j].c \leq t'[i].c] \\ & \wedge \ ((g'.loc = closing \wedge g'.c < 1) \vee (g'.loc = close \wedge g'.c \geq 1)) \\ \Rightarrow & n'_{L1} \vee n'_{L2} \end{array}$$

For the edge from n_{L2} to n_{L3} that represents the transition of train-i from control location *on* to *far*

$$\begin{array}{ll} & n_{L2} \wedge t' = [t \text{ EXCEPT } t[i]'.loc = far \wedge t[i]'.c < 20] \\ & \wedge \ g'.loc = close \wedge g'.c \geq 1 \\ & \wedge \ ctl'.loc = ctl.loc \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow & n'_{L2} \vee n'_{L3} \end{array}$$

For the edge from n_{L3} to n_{L4} that represents the set of transitions of train-j group from control location *on* to *far*

$$\begin{array}{ll} & n_{L3} \wedge t' = [t \text{ EXCEPT } t[j]'.loc = far \wedge t[j]'.c \leq t[i]'.c] \\ & \wedge \ g'.loc = opening \wedge g'.c \leq 1 \\ & \wedge \ ctl'.loc = free \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow & n'_{L3} \vee n'_{L4} \end{array}$$

For the edge from n_{L4} to n_{R1} that represents the set of transitions of train-j group from control location *far* to *near*.

$$\begin{array}{ll} & n_{L4} \wedge t' = [t \text{ EXCEPT } t[j]'.loc = near \wedge t[j]'.c < 6] \\ & \wedge \ g'.loc = open \wedge g'.c \leq 1 \\ & \wedge \ ctl'.loc = free \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow & n'_{L4} \vee n'_{R1} \end{array}$$

For the edge from n_{L2} to n_{R3} that represents the set of transitions of train-j group from control location *on* to *far* and $t[i]$ is in the same locations *near* or *on*

$$\begin{array}{ll} & n_{L2} \wedge t' = [t \text{ EXCEPT } t[j]'.loc = far \wedge t[j]'.c < 6] \\ & \wedge \ g'.loc = close \wedge g'.c \geq 1 \\ & \wedge \ ctl'.loc = ctl.loc \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow & n'_{L2} \vee n'_{R3} \end{array}$$

For the edge from n_{L4} to n_0 that represents the transition of train-i at control location *far*

$$\begin{array}{ll} & n_{L4} \wedge \forall i, j \in 1, \ldots, N \wedge i \neq j : t'[i] = far \wedge t'[i].c < 20 \\ & \wedge \ t'[j].loc = far \wedge t'[j].c < 20 \\ & \wedge \ g'.loc = open \wedge g'.c \leq 1 \\ & \wedge \ ctl'.loc = free \wedge ctl'.cnt \leq (N-1) \\ \Rightarrow & n'_{L4} \vee n'_0 \end{array}$$

CVC-Lite Codes

```
Process_train : TYPE = ARRAY INT OF [# loc: INT, c:REAL #];

%% to protect type checking errors

Process_controller : TYPE =  [# loc: INT, cnt:INT #];

Process_gate : TYPE =  [# loc: INT, c:REAL #];


t, t1 : Process_train ;

g, g1 : Process_gate ;

ctl, ctl1 : Process_controller ;

i, N : INT ;

N0_A1 : BOOLEAN = g.loc = 0 AND ctl.loc = 0
     AND (1 <= i AND i <= N)
     AND t[i].loc=0 AND t[i].c < 20 ;

N0_A2 : BOOLEAN = (1 <= i AND i <= N) AND
      FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
        AND t[j].loc=0 AND t[j].c < 20 ;

N0_AA : BOOLEAN = N0_A1 AND N0_A2 ;

N0_A11 : BOOLEAN = g1.loc = 0 AND ctl1.loc = 0
     AND (1 <= i AND i <= N)
     AND t1[i].loc=0 AND t1[i].c < 20 ;

N0_A21 : BOOLEAN = (1 <= i AND i <= N) AND
      FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
        AND t1[j].loc=0 AND t1[j].c < 20 ;

N0_AA1 : BOOLEAN = N0_A11 AND N0_A21 ;

%% Righthand %%

NR1_A1 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= (N-1))
          AND (ctl.loc = 0 OR ctl.loc = 1)
          AND (g.loc = 0 OR g.loc = 1)
```

```
          AND (1 <= i AND i <= N)
          AND t[i].loc=0 AND t[i].c < 20 ;

%NR1_A2 : BOOLEAN = (1 <= i AND i <= N) AND
%       FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
%          AND (t[j].loc=0 OR t[j].loc=1 OR t[j].loc=2) ;

NR1_A2 : BOOLEAN = (1 <= i AND i <= N) AND
      FORALL (j:INT): ((j /= i AND 1 <= j AND j <= N)
          AND (t[j].loc=0 OR t[j].loc=1)
          AND ctl.cnt <= (N-1)
          AND g.loc = 0 AND ctl.loc = 0)
       => (t[i].c >= t[j].c);

NR1_AA : BOOLEAN = NR1_A1 AND NR1_A2 ;

NR2_A1 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= N)
          AND (1 <= i AND i <= N)
          AND (t[i].loc=1 OR t[i].loc=2) ;

NR2_A2 : BOOLEAN = (1 <= i AND i <= N) AND
        FORALL (j:INT):(j /= i AND 1 <= j AND j <= N
         AND (t[j].loc=1 OR t[j].loc=2)
         AND g.loc=2
         AND ctl.cnt = N AND ctl.loc = 1)
        => (t[i].c <= t[j].c);

NR2_AA : BOOLEAN = NR2_A1 AND NR2_A2;

NR3_A1 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= (N-1))
          AND (1 <= i AND i <= N)
          AND (t[i].loc=1 OR t[i].loc=2) ;

NR3_A2 : BOOLEAN = (1 <= i AND i <= N) AND
       (FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
         AND (t[j].loc=0 OR t[j].loc=1)
         AND g.loc = 2
         AND ctl.cnt <= N AND ctl.loc = 1)
        => (g.c >= 1);

NR3_AA : BOOLEAN = NR3_A1 AND NR3_A2 ;

NR4_A1 : BOOLEAN = ( 0 <= ctl.cnt AND ctl.cnt <= (N-1))
          AND (1 <= i AND i <= N)
```

```
          AND t[i].loc=0 ;

NR4_A2 : BOOLEAN = (1 <= i AND i <= N) AND
       ( FORALL (j:INT):(j /= i AND 1 <= j AND j <= N)
         AND (t[j].loc=0 OR t[j].loc=1)
         AND g.loc = 3
         AND (ctl.cnt = 0 OR ctl.cnt = (N-1))
         AND ctl.loc = 0)
        => (g.c <= 1);

NR4_AA : BOOLEAN = NR4_A1 AND NR4_A2;

%% Righthand prime %%

NR1_A11 : BOOLEAN =  (0 <= ctl1.cnt AND ctl1.cnt <= (N-1))
          AND (ctl1.loc = 0 OR ctl1.loc = 1)
          AND (g1.loc = 0 OR g1.loc = 1)
          AND (1 <= i AND i <= N)
          AND t1[i].loc=0 AND t1[i].c < 20 ;

%NR1_A21 : BOOLEAN = (1 <= i AND i <= N) AND
%       FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
%          AND (t1[j].loc=0 OR t1[j].loc=1 OR t1[j].loc=2) ;

NR1_A21 : BOOLEAN = (1 <= i AND i <= N) AND
         FORALL (j:INT) : ((j /= i AND 1 <= j AND j <= N)
           AND (t1[j].loc=0 OR t1[j].loc=1)
           AND ctl1.cnt <= (N-1)
           AND g1.loc = 0 AND ctl1.loc = 0)
       => (t1[i].c >= t1[j].c);

NR1_AA1 : BOOLEAN = NR1_A11 AND NR1_A21 ;

NR2_A11 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= N)
          AND (1 <= i AND i <= N)
          AND (t1[i].loc=1 OR t1[i].loc=2) ;

NR2_A21 : BOOLEAN = (1 <= i AND i <= N) AND
        FORALL (j:INT):(j /= i AND 1 <= j AND j <= N
         AND (t1[j].loc=1 OR t1[j].loc=2)
         AND (g1.loc = 0 OR g1.loc=1)
         AND ctl1.cnt = N AND ctl1.loc = 1)
        => (t1[i].c <= t1[j].c);
```

```
NR2_AA1 : BOOLEAN = NR2_A11 AND NR2_A21;

NR3_A11 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= (N-1))
          AND (1 <= i AND i <= N)
          AND (t1[i].loc=1 OR t1[i].loc=2) ;

NR3_A21 : BOOLEAN = (1 <= i AND i <= N) AND
       (FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
         AND (t1[j].loc=0 OR t1[j].loc=1)
         AND g1.loc = 2
         AND ctl1.cnt <= N AND ctl1.loc = 1)
        => (g1.c >= 1);

NR3_AA1 : BOOLEAN = NR3_A11 AND NR3_A21;

NR4_A11 : BOOLEAN = ( 0<= ctl1.cnt AND ctl1.cnt <= (N-1))
          AND (1 <= i AND i <= N)
          AND t1[i].loc=0 ;

NR4_A21 : BOOLEAN = (1 <= i AND i <= N) AND
        (FORALL (j:INT):(j /= i AND 1 <= j AND j <= N)
         AND (t1[j].loc=0 OR t1[j].loc=1)
         AND g1.loc = 3
         AND (ctl1.cnt = 0 OR ctl1.cnt = (N-1))
         AND ctl1.loc = 0)
        => (g1.c <= 1);

NR4_AA1 : BOOLEAN = NR4_A11 AND NR4_A21;

%% Lefthand nodes %%

NL1_A1 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= (N-1))
          AND (ctl.loc = 0 OR ctl.loc = 1)
          AND (g.loc = 0 OR g.loc = 1)
          AND (1 <= i AND i <= N)
          AND t[i].loc=1 AND t[i].c < 6 ;

NL1_A2 : BOOLEAN = (1 <= i AND i <= N) AND
       FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
          AND (t[j].loc=0 OR t[j].loc=1 OR t[j].loc=2) ;

NL1_A3 : BOOLEAN = (1 <= i AND i <= N) AND
       FORALL (j:INT):(j /= i AND 1 <= j AND j <= N
        AND t[j].loc=1  AND ctl.cnt = 1
```

```
        AND g.loc = 0 AND ctl.loc = 0)
       => (t[i].c <= t[j].c);

NL1_AA : BOOLEAN = NL1_A1 AND NL1_A2 AND NL1_A3;

NL2_A1 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= N)
          AND (1 <= i AND i <= N)
          AND (t[i].loc=1 OR t[i].loc=2) ;

NL2_A2 : BOOLEAN = (1 <= i AND i <= N) AND
        FORALL (j:INT): (j /= i AND 1 <= j AND j <= N
           AND (t[j].loc=1 OR t[j].loc=2)
           AND g.loc=2
           AND ctl.cnt = N AND ctl.loc = 1)
        => (t[i].c >= t[j].c);

NL2_AA : BOOLEAN = NL2_A1 AND NL2_A2;


NL3_A1 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= (N-1))
          AND (1 <= i AND i <= N)
          AND t[i].loc=0 ;

NL3_A2 : BOOLEAN = (1 <= i AND i <= N) AND
       (FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
          AND (t[j].loc=1 OR t[j].loc=2)
          AND g.loc = 2
          AND ctl.cnt = (N-1) AND ctl.loc = 1)
        => (g.c >= 1);

NL3_AA : BOOLEAN = NL3_A1 AND NL3_A2;

NL4_A1 : BOOLEAN =  (0 <= ctl.cnt AND ctl.cnt <= (N-1))
          AND (1 <= i AND i <= N)
          AND (t[i].loc=0 OR t[i].loc=1) ;

NL4_A2 : BOOLEAN = (1 <= i AND i <= N) AND
        FORALL (j:INT):(j /= i AND 1 <= j AND j <= N)
         AND t[j].loc=0 AND g.loc = 3
         AND (ctl.cnt = 0 OR ctl.cnt = 1)
         AND ctl.loc = 3
        => (g.c <= 1);

NL4_AA : BOOLEAN = NL4_A1 AND NL4_A2;
```

```
%% Lefthand nodes primes%%

NL1_A11 : BOOLEAN =  (0 <= ctl1.cnt AND ctl1.cnt <= (N-1))
          AND (ctl1.loc = 0 OR ctl1.loc = 1)
          AND (g1.loc = 0 OR g1.loc = 1)
          AND (1 <= i AND i <= N)
          AND t1[i].loc=1 AND t1[i].c < 6 ;

NL1_A21 : BOOLEAN = (1 <= i AND i <= N) AND
       FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
          AND (t1[j].loc=0 OR t1[j].loc=1 OR t1[j].loc=2) ;

NL1_A31 : BOOLEAN = (1 <= i AND i <= N) AND
       FORALL (j:INT):(j /= i AND 1 <= j AND j <= N
        AND t1[j].loc=1  AND ctl1.cnt = 1
        AND g1.loc = 0 AND ctl1.loc = 0)
       => (t1[i].c <= t1[j].c);

NL1_AA1 : BOOLEAN = NL1_A11 AND NL1_A21 AND NL1_A31;

NL2_A11 : BOOLEAN =  (0 <= ctl1.cnt AND ctl1.cnt <= N)
          AND (1 <= i AND i <= N)
          AND (t1[i].loc=1 OR t1[i].loc=2) ;

NL2_A21 : BOOLEAN = (1 <= i AND i <= N) AND
        FORALL (j:INT): (j /= i AND 1 <= j AND j <= N
           AND (t1[j].loc=1 OR t1[j].loc=2)
           AND (g1.loc = 1 OR g1.loc=2)
           AND ctl1.cnt = N AND ctl1.loc = 1)
        => (t1[i].c >= t1[j].c);

NL2_AA1 : BOOLEAN = NL2_A11 AND NL2_A21;

NL3_A11 : BOOLEAN =  (0 <= ctl1.cnt AND ctl1.cnt <= (N-1))
          AND (1 <= i AND i <= N)
          AND t1[i].loc=0 ;

NL3_A21 : BOOLEAN = (1 <= i AND i <= N) AND
       (FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
          AND (t1[j].loc=1 OR t1[j].loc=2)
          AND g1.loc = 2
          AND ctl1.cnt = (N-1) AND ctl1.loc = 1)
        => (g1.c >= 1);
```

```
NL3_AA1 : BOOLEAN = NL3_A11 AND NL3_A21;

NL4_A11 : BOOLEAN =  (0 <= ctl1.cnt AND ctl1.cnt <= (N-1))
          AND (1 <= i AND i <= N)
          AND (t1[i].loc=0 OR t1[i].loc=1) ;

NL4_A21 : BOOLEAN = (1 <= i AND i <= N) AND
        FORALL (j:INT):(j /= i AND 1 <= j AND j <= N)
         AND t1[j].loc=0 AND g1.loc = 3
         AND (ctl1.cnt = 0 OR ctl1.cnt = 1)
         AND ctl1.loc = 3
        => (g1.c <= 1);

NL4_AA1 : BOOLEAN = NL4_A11 AND NL4_A21;

%%% XTGs %%%

Init : BOOLEAN = ctl.cnt=0 AND ctl.loc=0 AND g.loc=0
     AND (1 <= i AND i <= N )
     AND t[i].loc=0 AND t[i].c < 20 AND
     FORALL (j:INT) : j /= i AND (1 <= j AND j <= N)
        AND t[j].loc = 0 AND t[j].c < 20;

%%% Init %%%
PUSH;
ASSERT  Init;
QUERY N0_AA;
POP;

%%% N0 -> NR1 (far_j -> near_j)
PUSH;
ASSERT N0_AA AND g1.loc=1 AND g1.c < 1
       AND ctl1.loc=0 AND ctl1.cnt <= (N-1)
       AND (1 <= i AND i <= N) AND
    (EXISTS (j:INT):(j /= i AND 1 <= j AND j <= N)
     => ( FORALL (k:INT) : (k /= j AND 1 <= k AND k <= N)
      => (t1[k] = t[k] AND t1[j].loc=1 AND t1[j].c < 6)));

QUERY NR1_AA1;
POP;

%%% NR1 -> NR2
%%% (near_j -> on_j) and (far_i -> near_i)
```

```
%%% (near_j -> on_j) and (near_i -> on_i)
%PUSH;
%ASSERT NR1_AA AND ctl1.loc=1 AND ctl1.cnt <= N
%       AND ((g1.loc=1 AND g1.c < 1) OR (g1.loc=2 AND g1.c >= 1))
%       AND (1 <= i AND i <= N) AND
%      (EXISTS (j:INT):(j /= i AND 1 <= j AND j <= N)
%       => ( FORALL (k:INT) : (k /= j AND 1 <= k AND k <= N)
%         => (t1[k] = t[k] AND t1[j].loc=2 AND
%             (t1[i].loc = 1 OR t1[i].loc = 2) AND
%             t1[i].c >= t1[j].c )));

%QUERY NR2_AA1;
%POP;

%Unknown.
%CVC Lite was incomplete in this example due to:
% * Quantifier instantiation

%%% NR2 -> NR3 (on_j -> far_j) %%%
%PUSH;
%ASSERT NR2_AA AND g1.loc=2 AND g1.c >= 1
%       AND ctl1.loc=ctl.loc AND ctl1.cnt <= (N-1)
%       AND (1 <= i AND i <= N) AND
%       (EXISTS (j:INT):(j /= i AND 1 <= j AND j <= N)
%       => (FORALL (k:INT) : (k /= j AND 1 <= k AND k <= N)
%         => (t1[k] = t[k] AND t1[j].loc=0
%             AND t1[j].c < 20)));

%QUERY NR3_AA1;
%POP;
%CVC Lite was incomplete in this example due to:
% * Quantifier instantiation

%%% NR3 -> NR4
%%% (far_j -> far_j) and (on_i -> far_i)
PUSH;
ASSERT NR3_AA AND ctl1.loc=0 AND ctl1.cnt <= (N-1)
       AND g1.loc=3 AND g1.c < 1
       AND (1 <= i AND i <= N) AND t1[i].loc=t[i].loc AND
     (FORALL (j:INT) : j /= i AND (1 <= j AND j <= N)
       AND t1[j].loc = 0 AND t1[i].c <= t1[j].c) ;
QUERY NR4_AA1;
POP;
```

```
%%% NR2 -> NL3
%%% (on_i -> far_i) and (on_j -> on_j)
%PUSH;
%ASSERT NR2_AA AND ctl1.loc=ctl.loc AND ctl1.cnt <= (N-1)
%       AND g1.loc=2 AND g1.c >= 1
%       AND (1 <= i AND i <= N) AND
%     (FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
%      => (t1[j]=t[j] AND t1[i].loc = 0 AND t1[i].c < 20)) ;
%QUERY NL3_AA1;
%POP;
%CVC Lite was incomplete in this example due to:
% * Quantifier instantiation


%%% NR4 -> NR0
PUSH;
ASSERT NR4_AA AND ctl1.loc=0 AND ctl1.cnt = 0
       AND g1.loc=0 AND g1.c < 1
       AND (1 <= i AND i <= N)
       AND t1[i].loc=0 AND t1[i].c < 20 AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
       AND t1[j].loc = 0 AND t1[j].c < 20 ;
QUERY N0_AA1;
POP;

%% NR4 -> NL1 (far_i -> near_i) %%
%PUSH;
%ASSERT NR4_AA AND ctl1.loc=0 AND ctl1.cnt <= (N-1)
%       AND g1.loc=0 AND g1.c < 1
%       AND (1 <= i AND i <= N) AND
%      (FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
%       => (t1[j]=t[j] AND t1[i].loc = 1 AND t1[i].c < 6));
%QUERY NL1_AA1;
%POP;
%CVC Lite was incomplete in this example due to:
% * Quantifier instantiation

%% N0 -> NL1 (far_i -> near_i) %%
PUSH;
ASSERT N0_AA AND ctl1.loc=ctl.loc AND ctl1.cnt = 1
       AND g1.loc=g.loc AND g1.c < 1
       AND (1 <= i AND i <= N) AND
      FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
       => (t1[j]=t[j] AND t1[i].loc = 1 AND t1[i].c <= t1[j].c);
```

```
QUERY NL1_AA1;
POP;

%%% NL1 -> NL2
%%% (near_i -> on_i) and ((far_j -> near_j) or (near_j -> on_j))
PUSH;
ASSERT NL1_AA AND ctl1.loc=1 AND ctl1.cnt <= N
       AND ((g1.loc=1 AND g1.c < 1) OR (g1.loc=2 AND g1.c >= 1))
       AND (1 <= i AND i <= N) AND
      (EXISTS (j:INT):(j /= i AND 1 <= j AND j <= N)
       => ( FORALL (k:INT) : (k /= j AND 1 <= k AND k <= N)
         => (t1[k] = t[k] AND t1[i].loc=2 AND
             (t1[j].loc = 1 OR t1[j].loc = 2) AND
             t1[i].c <= t1[j].c )));

QUERY NL2_AA1;
POP;

%%% NL2 -> NL3 (on_i -> far_i) %%%
%PUSH;
%ASSERT NL2_AA AND g1.loc=2 AND g1.c >= 1
%       AND ctl1.loc=ctl.loc AND ctl1.cnt <= (N-1)
%       AND (1 <= i AND i <= N) AND
%       (EXISTS (j:INT):(j /= i AND 1 <= j AND j <= N)
%       => (FORALL (k:INT) : (k /= j AND 1 <= k AND k <= N)
%         => (t1[k] = t[k] AND t1[i].loc=0
%             AND t1[i].c < 20)));

%QUERY NL3_AA1;
%POP;
%CVC Lite was incomplete in this example due to:
% * Quantifier instantiation

%%% NL2 -> NR3
%PUSH;
%ASSERT NL2_AA AND g1.loc=2 AND g1.c >= 1
%       AND ctl1.loc=ctl.loc AND ctl1.cnt <= (N-1)
%       AND (1 <= i AND i <= N) AND
%       (EXISTS (j:INT):(j /= i AND 1 <= j AND j <= N)
%       => (FORALL (k:INT) : (k /= j AND 1 <= k AND k <= N)
%         => (t1[k] = t[k] AND t1[j].loc=0
%             AND t1[j].c < 20)));

%QUERY NR3_AA1;
```

```
%POP;
%CVC Lite was incomplete in this example due to:
% * Quantifier instantiation

%%% NL3 -> NL4
%%% (far_i -> far_i) and (on_j -> far_j)
PUSH;
ASSERT NL3_AA AND ctl1.loc=0 AND ctl1.cnt <= (N-1)
       AND g1.loc=3 AND g1.c < 1
       AND (1 <= i AND i <= N) AND t1[i].loc = t[i].loc AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
       AND t1[j].loc = 0 AND t1[i].c >= t1[j].c ;
QUERY NL4_AA1;
POP;

%%% NL4 -> N0
PUSH;
ASSERT NL4_AA AND ctl1.loc=0 AND ctl1.cnt = 0
       AND g1.loc=0 AND g1.c < 1
       AND (1 <= i AND i <= N)
       AND t1[i].loc=0 AND t1[i].c < 20 AND
     FORALL (j:INT) : (j /= i AND 1 <= j AND j <= N)
       AND t1[j].loc = 0 AND t1[j].c < 20 ;

QUERY N0_AA1;
POP;

%% NL4 -> NR1 (far_j -> near_j) %%
%PUSH;
%ASSERT NL4_AA AND ctl1.loc=0 AND ctl1.cnt <= (N-1)
%      AND g1.loc=0 AND g1.c < 1
%      AND (1 <= i AND i <= N) AND
%   (EXISTS (j:INT):(j /= i AND 1 <= j AND j <= N)
%     => (FORALL (k:INT) : (k /= j AND 1 <= k AND k <= N)
%       => (t1[k] = t[k] AND t1[j].loc=1)));
%QUERY NR1_AA1;
%POP;
%CVC Lite was incomplete in this example due to:
% * Quantifier instantiation

%% Property Verification %%
% N0 - NR4 - NL4 - NL3
PUSH;
ASSERT N0_AA OR NL3_AA OR NR4_AA OR NL4_AA ;
```

```
QUERY (t[i].loc = 2) => (g.loc=2);
POP;

%% NR3 - NR2 - NL2
PUSH;
ASSERT (t[i].loc= 1 OR t[i].loc=2) AND g.loc=2 ;
QUERY (t[i].loc = 2) => (g.loc=2) ;
POP;

%% NR1 - NL1
PUSH;
ASSERT (t[i].loc = 0 OR t[i].loc= 1) AND
       (g.loc = 0 OR g.loc=1);
QUERY  (t[i].loc = 2) => (g.loc=2);
POP;
```

Bibliography

[1] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhench. Verification of infinite-state systems by combining abstraction and reachability analysis. In *Proceedings 11th International Conference on Computer Aided Verification, CAV'99*, volume 1633 of *Lecture Notes in Computer Science*, pages 146–159. Springer-Verlag, 1999.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Ambridge University Press, 1996.

[3] R. Alur and D. Dill. The theory of timed automata. *Theoretical Computer Science*, (126):183–235, 1994.

[4] Hasan Amjad. Combining model checking and theorem proving. Technical report, UCAM-CL-TR-601, ISSN 146-2986, Cambridge University, pages 15-16, September 2004.

[5] M. Ammerlaan, R. Lutje Spelberg, and W.J. Toetenel. XTG – an engineering approach to modelling and analysis of real-time systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 88–97. IEEE press, 1998.

[6] Tobias Amnell and many others. UPPAAL: Now, next, and future. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, LNCS(2067):99-124. Springer-Verlag, Berlin, 2001.

[7] K. Apt and D. Kozen. *Limits for Automatic verification of finite-state concurrent systems.* Information Process letters, Vol 15, 1986.

[8] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In *Proceedings of the 1st*

International Workshop on Hybrid and Real-Time Systems, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, 1997.

[9] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the Programming Language Design and Implementation*, pages 203–213. ACM Press, 2001.

[10] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages (POPL 2002)*, pages 1–3, 2002.

[11] Thomas Ball and Rajamani Sriram K. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM Press, 2002.

[12] Giosué Bandini, Ronald Lutje Spelberg, and Hans Toetenel. Parametric model-checking in pmc.

[13] K. Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In *Proceedings of the 6th Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 188–204. Springer-Verlag, 2000.

[14] Kai Baukus, Yassine Lakhnech, and Larsten Stahl. Verifying universal properties of parameterized networks. Technical report, Technical report TR-sT-00-4, CAU Kiel, July 2000.

[15] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In *Proceedings 10th International Conference on Computer Aided Verification, CAV'98*, 505-510. Springer-Verlag, 1998.

[16] Dominique Cansell and Dominique Mery. Tutorial on the event-based B method. Technical report, LORIA-INRIA, 2004.

[17] Dominique Cansell, Dominique Mery, and Stephan Merz. Predicates diagrams for the verification of reactive systems. In *Proceedings the 2nd International Conference on Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[18] Dominique Cansell, Dominique Mery, and Stephan Merz. Diagram refinements for the design of reactive systems. *Journal of Universal Computer Science,7(2):159-174*, 2001.

[19] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings 12th International Conference on Computer Aided Verification, CAV'00*, 154-169. Springer-Verlag, 2000.

[20] E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the 6th annual ACM symposium on Principles of Distributed Computing*, pages 294–303. ACM Press, 1987.

[21] Michael Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304, 1998.

[22] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[23] CVC-Lite Homepage. Available at. http://www.cs.nyu.edu/acsys/cvcl.

[24] D. Dams. *Abstract Interpretation and Parition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[25] Satyaki Das. *Predicate Abstraction*. PhD thesis, Stanford University, 2004.

[26] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In *7th CAV95, LNCS(939):409–422*. Springer-Verlag, 1995.

[27] Loic Fejoz, Dominique Méry, and Stephan Merz. DIXIT: a graphical toolkit for predicate abstractions. In R. Bharadwaj and S. Mukhopadhyay, editors, *Intl. Workshop Automatic Verification of Infinite-State Systems (AVIS 2005)*, pages 39–48. LFCS, Univ. of Edinburgh, 2005. see also `http://www.loria.fr/equipes/mosel/dixit`.

[28] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2002.

[29] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On hte temporal analysis of fairness. In *ACM Symposium on principles of Programming Languages*, pages 163–173, 1980.

[30] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings 9th International Conference on Computer Aided Verification, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[31] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *9th CAV97, LNCS(1254):72–83*. Springer-Verlag, 1997.

[32] N. Halbwachs. Delay analysis in synchronous programs. In *CAV93, LNCS(697)*. Springer-Verlag, 1993.

[33] K. Havelund and N. Shankar. Experimeents in theorem providing and model checking for protocol verification. In *FME*, Lecture Notes in Computer Science, pages 662–681. Springer-Verlag, 1996.

[34] T.A. Henzinger and O. Kupferman. From quantity to quality. In *Proceedings of the 1st International Workshop on Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[35] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: The cornell hybrid technology tool. In *In proceedings of the 1st Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 29–43. Springer-Verlag, 1995.

[36] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth McMillan. Abstractions from proofs. In *31st Annual Symp. Princ. of Prog. Lang. (POPL 2004)*. ACM Press, 2004.

[37] Thomas A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. information and computation. In *Information and Computation*, volume 111(2):193-244, 1994.

[38] Eun-Young Kang. Parametric analysis of real-time embedded systems with abstract approximation interpretation. In *26th International Conference on Software Engineering*, pages 39–41, 2004.

[39] Eun-Young Kang. Real-time system verification techniques based on abstraction/deduction and model checking. In *Proceedings the 5th International Conference on Integrated Formal Methods*, CS-Report 05-29, pages 26–32. Technische Universiteit Eindhoven, 2005.

[40] Eun-Young Kang and Stephan Merz. Predicate diagrams for the verification of real-time system. In *5th International Workshop of Automated Verification of Critical Systems (AVoCS05), ENTCS(145):151–165*. Elsevier, 2005.

[41] Eun-Young Kang and Stephan Merz. Predicate diagrams for the verification of real-time systems. *Formal Aspects of Computing Journal*, 19(3):401–413, 2007.

[42] K.J. Kristoffersen, F. Laroussinie, K.G. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. Technical report, BRICS, Aalborg University, Denmark, 1996.

[43] K.G. Larsen, P. Petterson, and W. Yi. UPPAAL: Status and developments. Technical Report BRICS, Aalborg University, Denmark, 1997.

[44] David Lesens, Nicolas Halbwachs, and Pascal Raymond. Automatic verification of parameterized networks of processes. *Theoretical Computer Science*, 256(1–2):113–144, 2001.

[45] R.F. Lutje Spelberg, W.J. Toetenel, and M. Ammerlaan. Partition refinement in real-time model checking. In *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, 1998.

[46] Z. Manna and A. Pnueli. Verification of parameterized programs, 1995.

[47] Zohar Manna and Amir Pnueli. *Verification of parameterized programs: Specification and Validation Methods*. Oxford Univ, 1994.

[48] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems:safety*. Springer-Verlag, 1995.

[49] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[50] J. Misra and K.M. Chandy. *Parallel program design: a foundation*. Addison-Wesley Publishers, 1998.

[51] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer Aided Verification*, pages 435–449, 2000.

[52] Cecilia Esti Nugraheni. *Predicate diagrams as Basis for the verification of reactive systems*. PhD thesis, Munchen University, 2004.

[53] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[54] H. Sa. Modular and incremental analysis of concurrent software systems, 1999.

[55] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454, Trento, Italy, July 1999. Springer-Verlag.

[56] Ronald Lutje Spelberg. *Model Checking Real-Time Systems based on partition refinement*. PhD thesis, Delft University, 2004.

[57] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *Proceedings of the Eighth International Conference on CAV*, volume 1102, pages 232–243. Springer-Verlag, 1996.

[58] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science, pages 68–80. Springer-Verlag, 1990.

[59] Y.Kesten and A.Pnueli. Modularization and abstraction: The keys to practical formal verification. In *Proceedings of the 23th International Sumposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, 1998.

[60] Y.Kesten and A.Pnueli. Modularization and abstraction: The keys to practical formal verification. In *23th MFCS98, LNCS(1450):54-71.* Springer-Verlag, 1998.

[61] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1997.