N° d'ordre : 4641

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Farès Chucri**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Exploiting Model Structure in CEGAR Verification Method

Soutenue le : 27 Novembre 2012

Après avis des rapporteurs :

M. Jean-Michel Couvreur	Professeur, Université d'Orléans
Mme. Susanne Graf	Directrice de Recherche CNRS, VERIMAG

Devant la commission d'examen composée de :

M. Frédéric Boniol	Professeur, INP Toulouse	Président
M. Jean-Michel Couvreur	Professeur, Université d'Orléans	Examinateur
Mme. Susanne Graf	Directeur de Recherche CNRS, VERIMAG	Examinateur
M. Alain Griffault	MdC, Université Bordeaux 1	Co-directeur de thèse
M. Grégoire Sutre	Chargé de Recherche CNRS, LaBRI	Co-directeur de thèse
M. Igor Walukiewicz	Directeur de Recherche CNRS, LaBRI	Directeur de thèse

– 2012 –

Remerciement

Je souhaite tout d'abord remercier mes encadrants Igor Walukiewicz, Grégoire Sutre et Alain Griffault de m'avoir offert la chance unique de travailler à leurs côtés durant ces années. Merci de m'avoir soutenu, conseillé, et guidé à travers le dur parcours d'un doctorant. Merci encore pour ces encouragements et conseils qui m'ont permis de terminer ce travail ces deux dernières années malgré mon éloignement du LaBRI.

Merci au président du jury Frédéric Boniol, et aux rapporteurs M. Jean-Michel Couvreur et Susanne Graf.

Je remercie également toute l'équipe MF/MV du LaBRI pour son accueil chaleureux. Je souhaite tout particulièrement remercier Aymeric Vincent, pour les innombrables discutions, son aide dans le développement dans Mec 5, et son accueil lors de mes séjours à Bordeaux, et Emmanuel Fleury pour sa gentillesse.

Je remercie également ENSEIRB/IPB de m'avoir accueilli en tant que moniteur d'initiation à l'enseignement supérieur tout particulièrement Denis Lapoire, Frédéric Herbreteau, et l'ensemble du département d'informatique de m'avoir offert l'opportunité de découvrir l'enseignement universitaire.

Je remercie également ma compagne Christine Borowy pour son soutient et ses encouragements qui m'ont permis de trouver la motivation et l'énergie durant toutes ces années. Je remercie mes amis Jérôme Labeyrie et Maxime Bedexagar pour leur encouragement et accueil.

Je remercie ma famille et tout particulierment ma mère de m'avoir amené là où je suis, et surtout de m'avoir donné l'opportunité d'étudier toutes ces années.

Abstract

Software is now a key component of majority of devices and it is responsible for their safety and reliability. By safety we mean that the system must ensure that "bad things never happen". This type of property can be seen as a reachability problem: to prove the property, it suffices to prove that states designated as "bad" cannot be reached. This is particularly important for critical systems: systems whose failure can jeopardize human life, or economic liability.

We present two verifcation methods for AltaRicas models. First, a CEGAR algorithm that prunes away abstract states and therefore uses an underapproximation of the system state space is proposed. The use of our underapproximation of the abstract state space allow us to accelerate the algorithm. With our framework, we can pinpoint obvious feasible counterexamples, use reductions techniques to discard useless abstract states, minimize the cost of counterexample analysis, and guide the exploration of the abstraction towards counterexamples that are more likely to be feasible. We have implemented this framework in the model checker Mec 5, and experimental results confirmed the expected improvements.

We also propose a CEGAR algorithm for a subset of the AltaRica language: we consider the situation where we want to apply CEGAR algorithm to a hierarchical transition system. We want to do this without calculating the semantics of the hierarchical system. We propose to use hierarchical abstractions where each component is abstracted independently despite the presence of priorities in the model. This has three advantages: an abstraction is represented in a succinct way, it is easy to verify if an abstract path is spurious, the abstraction reflects the logical structure of the system.

Finally, we present the implementation our prunning algorithm in Mec 5. Benchmarks on a set of academic models, and on a large industrial case study illustrate the expected gain of our algorithm.

Résumé

Les logiciels sont désormais un des composants essentiels des équipements modernes. Ils sont responsables de leur sûreté et fiabilité. Par sûreté, nous entendons que le système garantit que “rien de dangereux n’arrive jamais”. Ce type de propriété peut se réduire à un problème d’accessibilité: pour démontrer la propriété il suffit de démontrer qu’un ensemble d’états “dangereux” ne sont pas atteignables. Ceci est particulièrement important pour les systèmes critiques: les systèmes dont une défaillance peut mettre en jeu des vies humaines ou l’économie d’une entreprise.

Afin de garantir un niveau de confiance suffisant dans nos équipements modernes, un grand nombre de méthodes de vérification ont étaient proposées. Ici nous nous intéressons au model checking: une méthode formelle de vérification de système. L’utilisation de méthodes de model checking et de model checker permet d’améliorer les analyses de sécurité des systèmes critiques, car elles permettent de garantir l’absence de bug vis-à-vis des propriétés spécifiées. De plus, le model checking est une méthode automatique, ceci permet à des utilisateurs non-spécialistes d’utiliser ces outils. Ceci permet l’utilisation de cette méthode à une grande communauté d’utilisateur dans différents contextes industriels. Mais le problème de l’explosion combinatoire de l’espace des états reste une difficulté qui limite l’utilisation de cette méthode dans un contexte industriel.

Nous présentons deux méthodes de vérification de modèle AltaRica. La première méthode présente un algorithme CEGAR qui élague des états de l’abstraction, ce qui permet d’utiliser une sous-approximation de l’espace des états d’un système. Grâce à l’utilisation de cette sous-approximation, nous pouvons détecter des contre-exemples simples, utiliser des méthodes de réduction pour éliminer des états abstraits, ce qui nous permet de minimiser le coût de l’analyse des contre-exemples, et guider l’exploration de l’abstraction vers des contre-exemples qui sont plus pertinents. Nous avons développé cet algorithme dans le model checker Mec 5, et les expérimentations réalisées ont confirmé

les améliorations attendues.

AltaRica

Le projet AltaRica [AGPR00] a débuté en 1996 du désir de partenaires industriels (Dassault Aviation, Total Fina Elf, Schneider Electric, AIRBUS) et académiques (LaBRI et ONERA) de créer un lien entre les méthodes formelles, et les analyses du fonctionnement et du dysfonctionnement des systèmes, et de développer des outils qui permettent de modéliser ces systèmes. AltaRica a récemment été utilisé pour obtenir la certification du système de contrôle de commande du jet Falcon 7X. Le langage AltaRica permet également de décrire des systèmes dès leurs premières phases de conception. Les outils industriels d'analyse de modèle AltaRica tel que Safety Designer [Das] et Simfia [Sim] permettent d'analyser des modèles finis mais contenant des milliers de variables booléens.

Dans un modèle AltaRica chaque composant est un noeud (un automate à contraintes) qui décrit le comportement d'une partie du système. Un noeud AltaRica peut contenir des sous-noeuds (un ensemble de noeuds AltaRica) et interagir avec eux. Deux model checker ont été développés pour AltaRica: Mec 5 et Arc [GV04, Vin03]. Mec 5 utilise des Binary Decision Diagrams pour représenter l'ensemble des états ainsi que la relation de transition. Arc utilise une représentation explicite de l'espace des états ainsi que des Decision Diagrams.

Réduction d'Abstraction

CEGAR. **C**ounter**E**xample **G**uided **A**bstraction **R**efinement est une méthode très efficace de vérification de propriétés d'atteignabilité. Cette méthode est basée sur le raffinement automatique de l'abstraction du système que l'on veut vérifier(e.g., [CGJ+03, HJMS02, SG04]). Ceci permet en particulier d'éviter la construction de l'ensemble des états du système. Donc, on peut à priori éviter le problème de l'explosion des états.

L'algorithme CEGAR peut se résumer ainsi: A chaque itération une abstraction du système est analysée. Si l'abstraction satisfait la propriété alors l'algorithme s'arrête et retourne "modèle sûr". Sinon, un contre-exemple abstrait est exhibé, et est analysé sur le système concret. S'il est exécutable sur le système concret, alors l'algorithme s'arrête et retourne "modèle non sûr". Sinon, si le contre-exemple n'est pas exécutable l'abstraction est raffinée afin d'éliminer le contre-exemple et l'algorithme reprend. Cette étape de raffinement est complexe et dépend de l'abstraction utilisée.

L'analyse de contre-exemples abstraits ainsi que le raffinement d'abstractions requiert le calcul coûteux d'un ensemble d'états accessibles dans le modèle concret. Ici nous présentons une méthode qui permet d'augmenter l'abstraction avec des informations sur les états accessibles concrets pour améliorer l'algorithme CEGAR.

Afin de pouvoir réduire l'abstraction et d'accélérer l'algorithme CEGAR, nous introduisons la notion de pair certifié: une extension de l'abstraction existentielle classique où certains états d'abstraction peuvent être identifiés comme ne représentant que des états

concrets accessibles ou coaccessibles. La principale contribution ici est la méthode de réduction basée sur les approximations certifiées. Cette méthode permet d'identifier des états abstraits inutiles et de les éliminer de l'abstraction. Cette réduction de l'espace des états permet de réduire les ressources nécessaires (temps et mémoire) à la construction et l'exploration des abstractions. De plus, cette réduction permet de d'éviter des raffinements inutiles (ceci permet d'accélérer l'algorithme CEGAR), et concentre l'algorithme sur des contre-exemples plus judicieux. Les approximations certifiées ne sont pas conservatives dans le sens classique étant donné que leur états abstraits représentent une sous approximation de l'espace des états concrets. Malgré cela nous montrons que l'on peut utiliser les approximations certifiées dans un algorithme CEGAR de façon sûre.

Afin d'augmenter l'ensemble des états certifiés, nous proposons différentes méthodes. Une première méthode basée sur les must transitions [LT88, BKY05] permet de d'obtenir facilement des états certifiés. Nous proposons également des méthodes basées sur la méthode de raffinement, et l'analyse de contre-exemples abstraits. Nous montrons également que l'ordre des opérations d'extension des états certifiés que l'on propose dans l'algorithme est optimale.

L'implémentation de l'algorithme de réduction d'abstractions dans le model checker Mec 5 est présentée ainsi qu'une analyse des expérimentations sur des modèles académiques et un modèle industriel.

CEGAR Hiérarchique

Les systèmes de transition sont rarement décrits explicitement. Ils sont souvent représentés comme la composition parallèle de systèmes de transitions basiques. La représentation modulaire amène ce concept plus loin: elle permet d'appliquer cette composition de façon hiérarchique. La sémantique de tels systèmes hiérarchiques est un simple système de transition. Il est aisé de voir que la sémantique peut être exponentiellement plus grande que sa représentation hiérarchique. Ici nous présentons des méthodes d'abstractions de ces systèmes de transitions hiérarchiques. L'objectif est d'éviter de calculer la sémantique du système hiérarchique, et surtout de tirer avantage de cette représentation d'un système sous forme de modules afin de trouver de "bonnes" abstractions rapidement.

La représentation hiérarchique est basée sur la composition parallèle. Nous considérons le produit synchrone de systèmes de transitions avec des vecteurs de synchronisation [AN82]. Dans la version la plus simple du produit synchrone, le produit de deux systèmes peut effectuer une action si les deux systèmes le peuvent. Dans un produit synchronisé plus élaboré avec des vecteurs de synchronisation une action d'un système peut être synchronisée avec une action d'un ou de plusieurs autres systèmes. Cette extension permet un produit synchrone plus flexible, et très utile dans une représentation hiérarchique d'un système.

Les priorités sont une autre possibilité que l'on considère. Une relation de priorité est un ordre partiel sur les actions. Si deux actions a et b sont possibles à partir d'un état, mais que b est plus prioritaire que a, alors ce sera b qui sera exécutée. Autrement

dit, l'action a sera bloquée. Vu ainsi, les priorités sont relativement simples. Elles deviennent plus puissantes lorsqu'elles sont utilisées conjointement avec la composition parallèle. Supposons que les actions a et b requièrent une synchronisation pour pouvoir être exécutées: par exemple a est synchronisée avec une action c, et b avec une action d. Si les autres composants permettent d'effectuer l'action c mais pas l'action d alors la synchronisation de a avec c sera exécutée malgré que b soit plus prioritaire. Mais si les autres composants permettaient d'exécuter les actions c et d, alors seule la synchronisation b et d serait possible. Les priorités ont un aspect temporisé et arborescent: elles permettent de détecter qu'une action n'est pas possible. Elles sont donc très utiles pour la modélisation (nous pouvons simplifier la description du modèle grâce à elles). Mais elles posent problème lorsque l'ont veut appliquer une méthode CEGAR sur des systèmes hiérarchiques contenant des priorités. D'un côté elles ajoutent des transitions dans la sémantique et en même temps elles en éliminent d'autres.

Notre objectif est d'étendre la méthode CEGAR au système hiérarchique. La méthode la plus simple est de calculer la sémantique du système hiérarchique et d'y appliquer n'importe quel algorithme CEGAR. A cause de la taille de la sémantique ceci n'est pas toujours possible. Nous proposons d'appliquer un algorithme CEGAR sans calculer la sémantique du système hiérarchique. Nous allons même plus loin en utilisant une abstraction hiérarchique qui reprend la hiérarchie du système que l'on veut analyser. Cette abstraction nous permet d'abstraire chaque système de transition séparément. Ceci nous permet de représenter l'abstraction d'une manière succincte.

La première difficulté de cette approche est qu'en général la composition des abstractions des différents composants n'est pas une abstraction du système original. Nous montrons que la notion d'abstraction par couverture s'adapte bien lorsque le système hiérarchique ne contient pas de priorités. De plus, nous montrons que lorsque le système ne contient pas de priorités, il est aisé de vérifier un contre-exemple abstrait: il suffit de vérifier la projection du contre-exemple sur chacun des composants.

Lorsque le système hiérarchique contient des priorités, la situation est plus complexe. A cause de l'impact des priorités ils n'est pas évident de garantir que la composition d'abstractions reste une abstraction. Pour contrecarrer cette situation, nous introduisons le concept de neat cover abstraction. Nous montrerons que grâce à cette notion nous retrouvons toutes les propriétés des systèmes hiérarchiques sans priorités.

Contents

Chapter 1

Introduction

1.1 Context & Motivations

In our modern society, software is now everywhere, from coffee machines to satellites navigation systems. The software implemented in a device is usually in charge of its functional behavior: it pilots the hardware in order to perform the task associated to a given input. The omnipresence of software has been made possible thanks to advances in the microchip, and computer, industry. This also allowed to implement more and more complex functions to widen the possibilities of devices. Software is now a key component of majority of devices and it is responsible for their safety and reliability. By safety we mean that the system must ensure that "bad things never happen". This type of property can be seen as a reachability problem: to prove the property, it suffices to prove that states designated as "bad" cannot be reached. This is particularly important for critical systems: systems whose failure can jeopardize human life, or economic liability.

Due to the size and complexity of modern systems it is impossible to verify them by inspection or test. Modern systems are no longer the product of a few engineers that designed and implemented the entire system. They are the product of a large number of actors: system architects, development/integration engineers that can work in different countries with different methods and perspective. A modern system can be the result of successive evolution. It can also be the result of integration of various independent systems put together for a particular task. In such situations there is person or a team that pilots the entire development of the system. Often despite human effort it is impossible to completely grasp the behavior of such a multi-layered system. This makes its verification a particularly complex and tedious task that nevertheless needs to be done in order to ensure it safety.

Formal methods appeared as an answer to the need for a verification method capable to prove safety of systems. These methods are based on a mathematical approach of the problem verification. On the bright side, formal methods ensure completeness:

a property of a systems is said "valid/proved/..." if and only if the systems satisfies the property, and otherwise a counterexample is exhibited. Model checking is a formal method approach for this task. The model checking problem can be formulated as follows: Given a model M and a property φ does M satisfies φ classically denoted $M \models \varphi$. The model M is a formal representation of a system, it can be given in various formalism from automaton to computer programming languages. A model checking algorithm (or method) is a procedure that can *automatically* decide if $M \models \varphi$ by an exhaustive search of the system state space. If $M \not\models \varphi$ the algorithm can return a counterexample that refutes φ in M. One of the advantages of model checking is it automatic approach to the verification problem. But, this method suffers from the state explosion problem: even if each individual module of a system has a modest size, the overall system has a size that is exponential in the number of modules. This issue makes the exhaustive exploration of the state space close to impossible for large systems.

In order to ensure a sufficient level of confidence in our modern systems, a number of pragmatic approaches have been proposed. One of them is the "V-Model" software development process has been introduced. This process is decomposed into five successive steps: system requirements, system specification, system architecture, detailed conception, and implementation. Each of this step is verified using tests: Unitary tests for each implemented function, integration tests to verify the interactions between functions, and finally validation tests to verify the specification. Despite this well structure process, a system developed according to the "V-Model" the may yet contains bugs. This is the drawback of tests: they can be easily implemented and executed, but they cannot prove the absence of bugs: at best test can only prove their presence. This is particularly problematic for critical systems that must satisfy safety properties.

The use of model checking methods and tools (model-checkers) improves the safety analysis of critical systems, because it guarantees that a system is "bug free" for the the specified properties. Moreover, the automatic verification approach offered by the model checking methods, makes it possible to employ model checkers by non-specialists. This widens the possible community of users and contexts where formal methods can help the development of safe systems. Yet the state explosion problem limits and sometimes forbids the use of model checking in an industrial context.

Continued efforts of the academic and industrial communities allowed to widen the scope of application of formal methods. These efforts, allowed an on-growing adoption of formal methods in the industry: formal methods are now "strongly recommended" in the CENELEC EN 50128 railway European norm for the safety analysis and validation of railway equipments, and have been introduced in the new DO 178 C, airborne systems and equipment certification norm.

Motivations

While model checking techniques are gaining popularity and are being more and more adopted in the industry, it becomes crucial for model-checkers to manage larger and larger systems. Yet due to the state explosion problem this is a challenging task on the

technical and theorical levels. The main issue is to verify with a model checker systems composed of a great number of modules.

Over time model checking methods have evolved from explicit state model checking, to symbolic model checking, and more recently to abstraction-based model checking techniques. Each of these methods has been build upon its predecessor in order to scale up the capabilities of model checkers to manage larger and larger systems. The explicit model checking approach explores the system in an efficient way in order to verify a property by testing all possible behaviors. This method reaches its limit when the system's state space is too large to be represented. The symbolic approach deals with the state space and exploration problem by different approaches: binary decision diagrams have been used to represent the state space and the transition relation of a system. With this concise representation that permits the use of efficient algorithms for logical operations it is possible to explore the system using only BDDs. Another popular symbolic approach to model checking is the formula based methods. Theses methods explore the model in order to find a counterexample by transforming the system into a (large) formula that represents its transition relation. The model checking problem is then "reduced" to a satisfiability problem. Both of these methods try to explore the system in search for a counterexample. The abstraction based methods tackle with the model checking problem differently: instead of exploring the system to verify a property, the property is verified on an abstraction of the system. The abstraction of the system is a smaller system that behaves as the original system, but can also introduce new behaviors. The advantage of the use of an abstraction is the possibility to manipulate a coarser representation of the original system, but the new behaviors introduced by the abstraction can induce spurious counterexample to the property under verification. Yet, this approach scales up once again the possibilities of model checkers to verify large systems, even if a the model checking problem gains in complexity due to the spurious counterexamples.

Among the abstraction-based methods the most successful approach is the well know CEGAR (CounterExample Guided Abstraction Refinement) method. This method, as suggested by its name, refines abstractions using counterexamples. The abstraction is indeed refined automatically in order to eliminate spurious counterexample discovered during the verification of a property on the abstraction. With this verification scheme it is now possible to model check large programs and models. The CEGAR method is tailored to determine the reachability properties. This makes it also a suitable verification method for safety properties.

The AltaRica language is a system description language. It is a popular modeling language that permits description of a system from its early stages of design to its implementation. It allows for instance the description of a system with non deterministic behavior in a modular and hierarchical way. Moreover, in AltaRica the modules can communicate using different methods like synchronization or dataflow. This flexibility offered by the language makes it possible to verify using model checking techniques a system at different stages of development: from its early specifications to implementation.

The existing AltaRica model checkers ARC and Mec 5 are explicit and symbolic model checkers using decision diagrams and binary decision diagrams respectively. Implementing a CEGAR model checker for the AltaRica language is a challenging task, that allows current AltaRica model checker to scale up to industrial models. This thesis presents this CEGAR extension of AltaRica. It offers also improvements of the CEGAR method on its key structure: the abstraction.

1.2 Contributions

The abstraction and the refinement methods are key steps of any CEGAR model checker. The first contribution of this thesis are abstraction methods allowing to build a sound and complete CEGAR algorithm that can prune abstract states even if they belong to abstract counterexamples. The second contribution is an abstraction scheme allowing to soundly abstract hierarchical transition systems with priorities. We propose a CEGAR algorithm that can analyze abstract counterexamples by projecting them on each element of the hierarchy even in the presence of priorities. The third contribution, is the implementation of the abstraction pruning CEGAR algorithm. This implementation is evaluated on academic models and a large industrial model.

For verification of AltaRica models, we first show how to enrich an exploit reachability information that is already available during the execution of the classical CEGAR loop in order build our CEGAR algorithm with pruning, PCegar. This algorithm relies on the use of certified approximations, our first abstraction scheme, that permits sound use of under-approximations of a system state space. Similar approaches referred to as "slicing" methods have been proposed by Jhalka et al. in [JM05] and Brückner et al. in [BDFW08]. Compared to our method, these methods are syntax-based, whereas our pruning method is semantic-based, and can therefore be applied to a larger variety of models. Certified approximations take advantage of certified states (abstract states that represent only reachable or coreachable concrete states) to prune away abstract states. Doing so, permits the CEGAR algorithm to focus on factors of abstract counterexamples. This permits to verify in a single iteration of the loop a set of abstract counterexamples. This pruning has many benefits: the (useless) refinements of pruned abstract states is avoided, the abstract counterexamples are shorter and more likely to be feasible since they represent one or more abstract counterexample of the corresponding non-pruned abstraction. Certified state inference methods are proposed and integrated in the CEGAR loop in order to maximize the pruning. Must transitions are used in order to statically infer certified states. The conjoint use of must transitions and certified pairs allows to solve the reachability problem by testing a simple condition that subsumes the one proposed in [BKY05]. This algorithm has been implemented in Mec 5, and the expected benefits have been observed on a set of academic benchmark models, as well as on an industrial model.

The second contribution is a compositional CEGAR algorithm for hierarchical transition systems: the modular mechanism of the AltaRica language. The key issue in this

model is the use of priorities (as in AltaRica) in transition systems. The presence of priorities makes it difficult to obtain an abstraction of a system even if we abstract it component-wise. We first present our HierarchicalCegar algorithm for the case when there are no priorities in the hierarchical transition system. We show how to verify abstract counterexample efficiently in this case. The introduction of priorities in the hierarchical transition system generates many problems: abstracting separately each component does not guarantee to obtain an abstraction of the hierarchical transition system, and moreover, it is not even possible to verify abstract counterexample efficiently. A notion of *neat cover* is proposed to solve these issues. Neat covers are a particular type of abstraction that permits the use of the generic compositional CEGAR algorithm HierarchicalCegar even when the hierarchical transition system contains priorities. Moreover, thanks to neat covers it is even possible to efficiently verify abstract counterexamples by projecting them on each component of the hierarchy. In [COYC03] Chacki et al. proposed a compositional CEGAR algorithm for C programs. The algorithm verified concurrent C programs using two levels of abstractions in order to reduce the abstract state space to manipulate. The method proposed in this thesis applies to hierarchical systems that can define local priorities, whereas the CEGAR algorithm proposed [COYC03] verifies C program without the hierarchical setting and without priorities.

As the last contribution of this thesis, we present the implementation of PCegar algorithm in Mec 5, and compare it to the classical Cegar algorithm. The expected benefits of the use of the pruning steps are illustrated by the benchmarks and discussed. This CEGAR extension is now part of Mec 5 publicly available at [Mec10].

Outline

This thesis is organized as follows: Chapter 2 presents the AltaRica language. A review of the model checking techniques is given in Chapter 3. Chapter 4 presents our CEGAR with pruning method, and Chapter 5 presents our CEGAR algorithm for hierarchical transition systems. The implementation of PCegar and a detailed presentation of some benchmarks model is the focus of Chapter 6. The conclusion of this thesis is presented in Chapter 7.

1.3 Preliminary Definitions & Notations

Given a set A, we write $\mathcal{P}^+(A) = \mathcal{P}(A) \setminus \{\emptyset\}$ for the set of non-empty subsets of A. For a binary relation $R \subseteq A \times A$, we write $x\,R\,y$ when $(x, y) \in R$. We denote by R^* the *reflexive and transitive closure* of R, and we write R^{-1} for its *inverse*. Given a subset $B \subseteq A$, the *forward image* of B by R is defined as $R[B] = \{y \in A \mid \exists x \in B : x\,R\,y\}$.

1.3.1 Transition Systems & Labeled Transition Systems

Transition systems are a classical representation of a system semantics.

Definition 1.1. *A* transition system *is a 4-tuple* $S = \langle Q, \to, I, F \rangle$ *where* Q *is a set of* states, $\to \subseteq (Q \times Q)$ *is a* transition relation, $I \subseteq Q$ *is a set of* initial states, *and* $F \subseteq Q$ *is a set of* final states. *A* labeled transition system *is a 5-tuple* $S = \langle Q, \Sigma, \to, I, F \rangle$, *where likewise where* Q *is a set of* states, Σ *is an alphabet,* $\to \subseteq (Q \times \Sigma \times Q)$ *is a* transition relation, $I \subseteq Q$ *is a set of* initial states, *and* $F \subseteq Q$ *is a set of* final states.

Given a transition system $S = \langle Q, \to, I, F \rangle$, we define the classical functions $post_S$, pre_S, $post_S^*$ and pre_S^* from $\mathcal{P}^+(Q)$ to $\mathcal{P}^+(Q)$ by:

$$post_S(X) = (\to)[X] \qquad post_S^*(X) = \bigcup_{i \in \mathbb{N}} post_S^i(X)$$
$$pre_S(X) = (\to)^{-1}[X] \qquad pre_S^*(X) = \bigcup_{i \in \mathbb{N}} pre_S^i(X)$$

The safety verification problem we address here can be seen as the search for the existence of a particular path in a transition system. The following definition formalizes the notions of paths and runs for (labeled) transitions systems.

Definition 1.2. *A* path *in a transition system* S *is a non-empty finite sequence of states* $q_0, \ldots, q_n$ *such that* $q_i \to q_{i+1}$ *for all* $0 \leq i < n$. *A* run *is a path* $q_0, \ldots, q_n$ *with* $q_0 \in I$ *and* $q_n \in F$. *The set of all paths (resp. runs) of* S *is denoted by* $Path(S)$ *(resp.* $Run(S)$*). We extend the definition of paths and runs to labeled transition systems by replacing the* $q_i \to q_{i+1}$ *condition by* $q_i \xrightarrow{a_i} q_{i+1}$.

Definition 1.3. *A* word *over an alphabet* Σ, *is a sequence of letters of* Σ. *Given a labeled transition system* S, *a word* $w = a_0, a_1, \ldots, a_n$ *is* accepted *by* S *if there exists a run* $q_0, q_1, \ldots, q_n$ *such that for every* $i = 0, \ldots, n$ *we have* $q_i \xrightarrow{a_i} q_{i+1}$, *and moreover* $q_0 \in I$, $q_n \in F$.

The set of words accepted by a labeled transition system S is its *language* and is denoted by $\mathcal{L}(S)$.

Chapter 2

AltaRica

The AltaRica project [AGPR00] started in 1996 from the wish of industrial partners (Dassault Aviation, Total Fina Elf, Schneider Electric, AIRBUS) and academic researchers (LaBRI, and ONERA) to link formal methods, reliability, risk assessment, quantitative analysis of dysfunctions and the qualitative analysis of functional behaviors, and to build tools and methods for the modeling of systems. It have recently been successfully used to certify the Falcon 7X turbojet flight controls commands. The AltaRica formal language is also well-suited to the description of early design models. Industrial AltaRica models, such as the ones obtained with the commercial tools Safety Designer [Das] and Simfia [Sim], are finite-state, but may contain over thousand boolean variables. Each component is an AltaRica node (basically a constraint automata) that describes the behavior of a part of the system. An AltaRica node may contain sub-nodes (a set of AltaRica nodes) and interact with those nodes. Two model checkers have been developed for AltaRica: Mec 5, and Arc [GV04, Vin03]. Mec 5 uses Binary Decision Diagrams (BDD for short) to represent sets of states, whereas Arc works with an explicit state representation as well as with DDs.

2.1 The AltaRica Description Language

The AltaRica language allows to describe a system in terms of constraint automata called *nodes*. Nodes are composed hierarchically. The hierarchy is represented by a finite unordered tree.

The AltaRica language distinguishes itself from other popular description languages such as Lustre, Promela, and SMV. Lustre is a synchronous flow oriented language where processes are linked using data flows. Promela (PROcess MEta LAnguage) is more oriented toward protocol modelization. It allows one to model a set of processes that communicate through channels. SMV is parallel process oriented language, where one can define a set of "modules" that evolve synchronously, and without a hierarchical structure of the processes. In comparison AltaRica is a less specialized asynchronous

modeling language. It is well suited to model protocols, processes. Moreover, despite its genericity, the modeling of systems is relatively direct.

An AltaRica leaf node is basically a constraint automaton which is usually modeled as a tuple containing:

- A set of variables to define a set of configurations as the product of variables domain,
- a set of events to define the labels of transitions,
- a set of labeled and guarded transitions,
- an initial condition on variables,
- an assertion to constrain the set of configurations, and
- a priority relation between events to restrict the set of transitions.

In a hierarchical AltaRica setting, the following elements are added:

- A set of subnodes to define a hierarchy, and
- a set of synchronization vectors.

Also note that in a hierarchical AltaRica node, the assertion and initial condition of a node can refer to its subnode's variables. Doing so, the node restrains its subnode configurations. Also, in order to simplify modeling in the language, implicit objects and default values have been introduced in the language.

In this chapter, we will start by presenting briefly the language features and its semantic. The former will be presented more precisely in a second part of this chapter. We do not intend to give an in-depth presentation of the language, for a detailed presentation of the AltaRica language see [AGPR00, Poi00, Vin03].

2.1.1 Leaf AltaRica Nodes

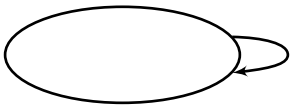
The Minimal Node

To begin our presentation of an AltaRica node, let us consider the simplest possible AltaRica node: The `Minimal` node. The AltaRica description of this node is given in Figure 2.1(a). This example allows us to introduce the first two basic keywords: **node** and **edon**. These keywords are the delimiters of an AltaRica node description. The description starts with the keyword **node** immediately followed by an identifier that is the name of the node. The description ends with the keyword **edon**.

Next to the AltaRica description of the node `Minimal` in Figure 2.1(b) we have depicted its semantic. Classically, the semantic is given as a transition system. Observe that despite the "emptiness" of the AltaRica description, in the semantic we yet have a state and a transition. This state is implicitly declared when an AltaRica description

```
node Minimal
edon
```

(a)



(b)

Figure 2.1: A simple AltaRica node: (a) the node Minimal, (b) the semantic of Minimal.

does not define any variable. It corresponds to an "idle" state of the AltaRica node. The transition that loops on the state is called an ε-transition, and likewise it can be seen as an "idle" transition. Note that ε-transitions are declared implicitly: they induce at least a loop on each state of the semantic. Note that as convention we will write ε label when drawing ε-transitions.

Now we can start building more complex nodes by introducing each element of an AltaRica node. In this chapter to illustrate the AltaRica language we will use as a running example a stack model.

States Variables & Initial Condition

An AltaRica node can manipulate two disjoint sets of variables: states, and flow variables.

State Variables. State variables as suggested by their name, describe the internal states of an AltaRica node. These variables can be seen as the internal (or local) variables of a node: they can be read and modified at will by the node. A state variable must be typed, and the AltaRica language predefines the following types:

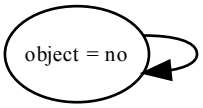
- *Boolean* using the keyword **bool**.
- *Integer* using the keyword **integer**.
- *Interval* using an interval definition of the form $[x, y]$, where x and y are integers.
- *Enumeration* using a set definition of the form $\{a, \ldots, z\}$.

Their declaration in an AltaRica description is preceded by the keyword **state**. Each variable is defined using an identifier followed by a colon and the type. They are given as a semicolon separated list. In Figure 2.2(a), we have our first stack cell given as the AltaRica node `Stack1`. A state variable *object* is defined and its type is an enumeration: $\{no, a, b\}$. This state variable represent the content of the stack: it is either empty when *object* is set to *no*, or contains an object a or b.

Initial Condition. An AltaRica node can specify an initial condition that define the initial states of its semantic. The initial condition is given as a list of semicolon separated assignments. The list is introduced by the keyword **init**.

```
node Stack1
  state object : {no, a, b};
  init  object := no;
edon
```

(a)



(b)

Figure 2.2: A stack cell: (a) the node Stack1, (b) the semantic of Stack1.

The `Stack1` AltaRica node of Figure 2.2(a) for example sets the state variable *object* to the value *no* as its initial condition. In Figure 2.2(b) we have the semantic of `Stack1`. Note that we only represent states that are reachable from the initial states, here the state where the value of *object* is *no*.

Flow Variables & Assertions

Flow Variables. Flow variables, like state variables can be defined in an AltaRica node. But these variables serve a different purpose: they usually represent the environment of an AltaRica node (or in some cases its parameters), and are an input and output interface with the environment as well. Flow variables cannot be modified directly by a node, they are constraint by the node and its environment. As state variables they must be typed using the same syntax. They are introduced by the keyword **flow** and are declared like state variables.

Flow variables are not free variables since with the help of an assertion one can constrain their values.

Assertion. An assertion is a semicolon separated list of boolean expressions that should be always satisfied by an AltaRica node. More formally, the assertion is the conjunction of the expressions. The assertion of an AltaRica node allows us to constrain the values of flow variables as well as state variables. Implicitly when no assertion is declared, its value is set to the truth value *true*. A valuation of state and flow variables that satisfy the assertion is called a *configuration* of a given AltaRica node.

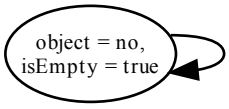
In Figure 2.3(a) we added to the `Stack1` node a flow boolean variable $isEmpty$. We want this variable to be *true* only when the cell does not hold an object. To this end, we define the assertion $isEmpty = (object = no)$. Note that the configuration associated to the state is its label.

Events & Transitions

We have described the static part of an AltaRica node. As we have seen there exist two types of variables: state variables, and flow variables. We also saw how to define an initial condition over the state variables, and to constrain flow variables. Now, we can turn our focus to the dynamic aspects of an AltaRica node, and go over events and transitions.

```
node Stack1
  state
    object : {no, a, b};
  flow
    isEmpty : bool;
  init
    object := no;
  assert
    isEmpty = ( object = no )
edon
```

(a)



(b)

Figure 2.3: A stack cell: (a) the node Stack1 extend with a flow variable, (b) the semantic of Stack1.

Events. A set of events can be specified for an AltaRica node. Events are primarily used to label transitions. They are introduced with the keyword **event**, and are given as a list of identifiers. Recall that the ε event is implicitly declared for all AltaRica nodes.

Going back to our running example, recall that a stack cell is a container that can hold elements. Here we model a stack cell that can contain two type of objects: an object of type a, and another one of type b. Classically, one can either push an element or pop an element that has been previously pushed. These actions are modeled in our AltaRica `Stack1` node in Figure 2.4(a) by the events *push*, and *pop*.

Now that we have events to model these actions, we need to define transitions that will do perform the desired actions.

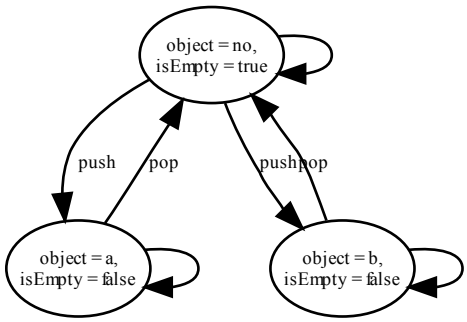
Transitions. We now have all the necessary ingredients to present AltaRica transitions. An AltaRica transition is a triplet made of a guard, an event, and an update. The *guard* is an expression over the variables of the node (state and flow variables). An event must label a transition. The *update* of a transition is a coma separated list of assignments. The updated variables can only be state variables, the update can be any arithmetic or boolean expression over the node variables (state and flow variables). When multiple updates are specified, the semantic imposes that they occur in parallel.

Given a configuration $\vec{c}$, a transition can be *fired* if and only if $\vec{c}$ satisfies the guard and there exists a configuration $\vec{c'}$ whose values reflect the application update of the transition with respect to $\vec{c}$. Note that we do not require the guards of different transitions to be mutually exclusive. This allows us to have AltaRica nodes that describe a non-deterministic system. Also note that when a transition is fired, non updated state variables can not have their value changed.

Syntactically, in an AltaRica node description transitions are introduced using the keyword **trans**. They are given as a semicolon separated list. A transition starts with a guard immediately followed by the base sign: $|-$. After the base, a non empty coma separated list of events must be declared. Each declared event will label the transition,

```
node Stack1
  state  object : {no, a, b};
  flow   isEmpty : bool;
  init   object := no;
  assert isEmpty = ( object = no )
  event push, pop;
  trans
  object = no |- push -> object := a;
  object = no |- push -> object := b;
  object = a |- pop -> object := no;
  object = b |- pop -> object := no;
edon
```

(a)



(b)

Figure 2.4: A complete stack cell: (a) the node Stack1 that model a simple stack cell, (b) the semantic of Stack1.

or put differently a copy of the transition will be added for each event. The event list is followed by the trans sign: ->. An optional coma separated list of updates can be declared at this point.

We have introduced the events *push* and *pop* to model the classical actions that a stack cell can do. Now we need to guarantee that we can push an object only when the stack cell is empty, and pop an object from an non-empty cell. With the help of transition, this is easily done.

In Figure 2.4(a) we have our final `Stack1` cell. Observe that we added transitions to manage the behavior of our stack cell. The first two transitions push an object in our stack. The guard of these transitions is $object = no$, the associated event is *push*, and the update assigns a or b to the *object* state variable. The guard guarantees that we can fire the transition labeled *push* only when the stack cell is empty: the state variable *object* has the value *no*. Note that these two transitions allow a non-deterministics choice: the *push* event inserts either an object a or b. The remaining two transitions allow us to pop out an object from the stack cell. Their guard guarantees that there is an object in the stack: the value of the variable *object* must be a or b. These transitions are naturally labeled with the *pop* event, and update the state variable *object* to the value *no* which empties the stack. Note, that we could have used the flow variable *isEmpty* as the guard of these transitions to factorize them into a single one by using the guard: $\sim isEmpty$. The semantic of our final `Stack1` node is given in Figure 2.4(b).

We have presented the local features of an AltaRica node. The AltaRica language allows to describe systems in a hierarchical manner. We now present the hierarchical aspects of AltaRica nodes.

2.1.2 The AltaRica Hierarchy

One of main advantages of AltaRica is its hierarchical description mechanism. In the following we will present the tree structure of an AltaRica node, and the interaction between a node an its subnodes.

Subnodes

An AltaRica node can define subnodes that are its *successors*, the node is their *predecessor*. Successors of an AltaRica node are called *siblings*. A node that does not define subnodes is called a *leaf* node. Subnodes are introduced by the keyword **sub**. They must be given as a semicolon separated list. A subnode declaration is a unique identifier that names the subnode followed by a colon, and a previously declared AltaRica node name. As in object oriented programming languages, an AltaRica node can be viewed as a generic object. A subnode declaration allows to instantiate a given AltaRica node. An illustration of an AltaRica node with two subnodes is given in Figure 2.5(a).

The AltaRica node `Stack2` of Figure 2.5(a) declares two subnodes namely *Top* and *Stack* as its subnodes, both are Stack1 nodes. Two siblings are independent: by default in an AltaRica hierarchy, only a single node can fire a non ε transition at each step. The set of global events can be viewed as the collection of the events of all the nodes (the node and its subnodes) together with a global ε event. To understand this, observe the semantic of `Stack2` given in Figure 2.6. As expected the semantic of `Stack2` is the product of the semantic of its subnodes with its own "local" semantic which is a single state transition system that can fire an ε labeled loop. Transitions are labeled with the events fired by the nodes, except that ε-transitions are not denoted explicitly. Observe that no transition is labeled with an event of both *Top* and *Stack* subnodes. In fact when a node fires a non ε-transition, the remaining nodes fire an ε-transition[1]. This restriction can be relaxed with the help of synchronization vectors [AN82]. We will present them bit later.

Yet our `Stack2` node does not behave like a proper stack. As it is defined for now, it is a simple container that holds two cells. This is due to the independence of our two `Stack1` subnodes. For instance we do not have control over the local *push* or *pop* events: they can occur in either the *Top* or *Stack* subnode. Another issue is the "user interface": we want our `Stack2` node to be the unique interface that masks the underlying system. Yet here in order to push or pop an object in our stack model the `Stack2` node must fire the ε transition.

Shared Variable Constraints

The AltaRica language allows us to define two types of interactions between a node and its subnodes. The first is shared variable constraints, and the second is synchronization of events.

[1] The ε event of each node serves as its "no operation" event and allows it to "wait".

```
node Stack2
  sub
    Top   : Stack1;
    Stack : Stack1;
edon
```

Figure 2.5: A basic two cell stack: the node Stack2.

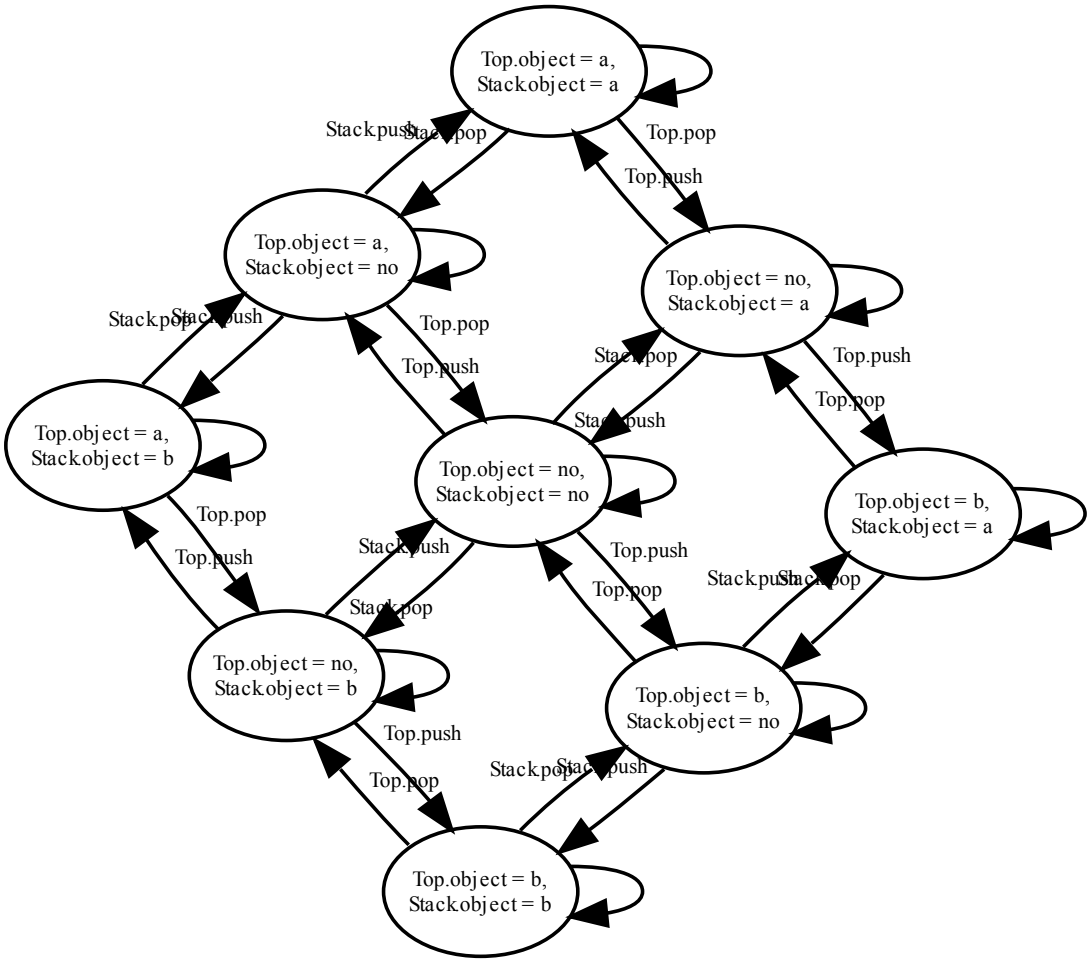


Figure 2.6: The semantic of Stack2.

```
node Stack2
  sub
    Top   : Stack1;
    Stack : Stack1;
  assert
    ~(Stack.object = no) | Top.object = no;
edon
```

Figure 2.7: A basic two cell stack with a shared variable constraint, (a) the node Stack2.

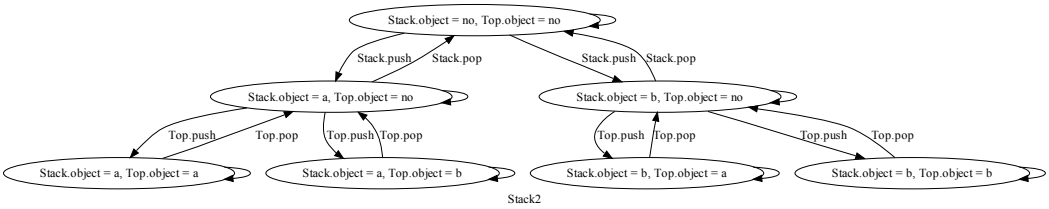


Figure 2.8: The semantic of Stack2.

A method to interact with a subnodes is *shared variables constraints*. An AltaRica node can use its subnodes flow variables in its assertion. Doing so, it can restrain the possible valuation (i.e. the configurations) of used subnodes variables. Moreover, we can use this to correlate subnodes variables between siblings, and/or between a node and its subnodes.

An illustration of the usage of subnode's variables in an assertion is given in Figure 2.7, and its semantic is given in Figure 2.8. In this example the node `Stack2` ensures that whenever the *Stack* subnode is empty the *Top* subnode is empty too. First, note that to refer to a subnode variable, we use a classical dot notation: the name of the subnode and its element we refer to are separated by a dot. More importantly, this simple assertion ensure a proper stack behavior of our container. Indeed, enforcing the emptiness of the *Top* subnode when the *Stack* subnode is empty guarantees that object will be inserted in a bottom up manner. In a way, this assertion is the specification of a proper stack model, and not a direct implementation. In the following, we will remodel our stack without this assertion using instead the order features offered by the AltaRica language. Moreover, even if this modelization is correct in terms of behavior, it still does not provide a proper user interface: from the `Stack2` node point of view we only fire ε transitions.

Synchronization

The AltaRica language implements another communication mechanism between a node and its subnodes: *synchronization vectors*. We have seen that by default all subnodes events are independent. However, we can specify *synchronization vectors* that link a node event with some of its subnodes events. An event that appear in a synchronization

```
node Stack2
  sub
    Top   : Stack1;
    Stack : Stack1;
  event
    pushT, pushS;
    popT, popS;
  trans
    true |- pushT, pushS, popT, popS -> ;
  sync
    <pushT, Top.push>;
    <pushS, Stack.push>;
    <popT, Top.pop>;
    <popS, Stack.pop>;
edon
```

Figure 2.9: The Stack2 extend with events and synchronization vectors.

vector is referred to as a *synchronized event*. A synchronized event can only occur[2] if all the events it is synchronized with can occur at the same time.

A synchronization vector defines a relation between the set of events of an AltaRica node and its subnodes. Theses vectors are introduced by the keyword **sync**. Synchronization vectors are given as a semicolon separated list. A synchronization vector is a coma separated list of events (local or subnodes events) enclosed within the "<" and ">" marks. A vector synchronizes one event from every subnode. By convention ε events are not written explicitly.

Thanks to synchronization vectors, we can now improve our `Stack2` node. First, we add four events: *pushT*, and *pushS* for "push Top" and "push Stack" respectively, and their dual events: *popT*, and *popS* for "pop Top" and "pop Stack" respectively. Then we synchronize these events with the subnodes *Top* and *Stack*. For instance we synchronize the *pushT* event with the *push* event of the subnode *Top*, and the *popT* event with the *pop* event of the *Top* subnode. Likewise we synchronize the *pushS* and *popS* with there counterparts of the *Stack* node. This `Stack2` node is given in Figure 2.9.

The modified `Stack2` node now behaves like a nice and practical container. The *pushT* and *pushS* events allow us to insert an object, and the *popT* and *popS* events allow us to pop the inserted objects. Since every event of `Stack2` and its subnodes appears in a synchronization vector, the set of global events is composed of the synchronization vectors together with a global ε event. The semantic of the `Stack2` node is given in Figure 2.10.

[2] We say that an event can occur, if a transition labeled with the event can be fired.

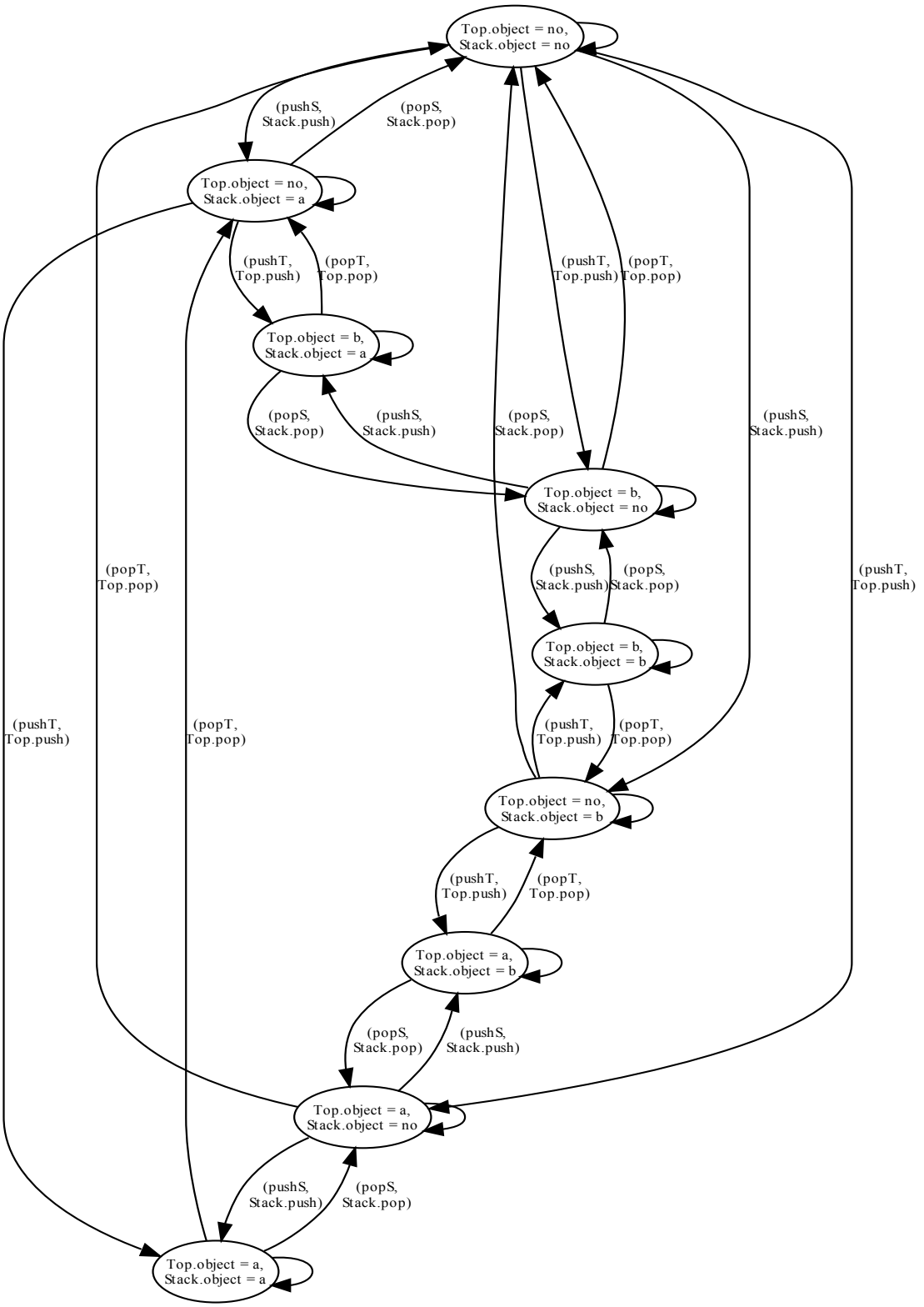


Figure 2.10: the semantic of Stack2.

Priorities

Our improved `Stack2` node still does not behave like a stack: we can for example pop an object from the tail of our stack (the subnode *Stack*) while having an object in the head (e.g. the transition from the state $(Top.object = b, Stack.object = a)$ to the state $(Top.object = b, Stack.object = no)$ labeled $(popS, Stack.pop)$ in Figure 2.10). In order to tackle this last issue, we now introduce priorities which are another facility offered by the AltaRica language.

With priorities we can use a "preferred action" mechanism. This allows us to eliminate transitions whose labels have smaller priority. Here, we want our `Stack2` container to push an object into its *Stack* subnode instead of its *Head* subnode whenever it can, and pop an object from its *Head* subnode instead of its *Stack* subnode whenever it can. More precisely we want to forbid (or eliminate) the *pushT* event whenever the *pushS* event is possible. Likewise we want to forbid the *popS* event when the *popT* event possible. This behavior can be easily implemented using priorities. With priorities we only need to give a higher priority to the *pushS* event over the *pushT* event. To do so, we can simply write $pushS > pushT$. Likewise, we also write $popT > popS$ to pop objects in the right manner.

More formally, in an AltaRica node, a partial order can be defined over the set of events. This partial order is called a *priority relation*. In an AltaRica node declaration, the priorities are introduced together with events. While defining events, using the less operator $<$ or greater operator $>$ as a separator we can define our priorities. Note that the partial order is not explicitly given. The priority relation associated to an AltaRica node description is the smallest partial order generated by the given pairs of priorities.

Priorities, operate at the semantic level: when a state has outgoing transitions labeled with comparable events, the transitions labeled with the events of lower priority are eliminated. Therefore, the priority relation can be viewed as a transition's "elimination" method. An important point about priorities is the moment of their evaluation: as we have seen they operate on the semantic level. In a leaf node predicting their impact on the transition is easy. But in a hierarchical node, some transitions are eliminated beforehand due to synchronizations, so the impact of priorities is harder to predict without computing all possible synchronizations.

Now let us go back to our `Stack2` node. In our last improvement, we have seen that `Stack2` can push or pop from any location of the stack. With the help of priorities we can now finalize our AltaRica model of a stack. The undesired behaviors we want to get rid of are the following:

1. Do not insert in the head if the stack is empty.
2. Do not pop from the stack if the head contains an object.

To deal with the first point we only need to specify that inserting in the stack has a higher priority than inserting in the head. The second issue is solved using the same scheme. The modified AltaRica description of `Stack2` is given in Figure 2.11.

```
node Stack2
  sub
    Top   : Stack1;
    Stack : Stack1;
  event
    pushT < pushS;
    popT > popS;
  trans
    true |– pushT, pushS,
             popT, popS –> ;
  sync
    <pushT, Top.push>;
    <pushS, Stack.push>;
    <popT, Top.pop>;
    <popS, Stack.pop>;
edon
```

Figure 2.11: An AltaRica model of a Stack of two elements, the node Stack2.

Now our stack model is correct: new objects are inserted bottom-up, and objects are popped in a top-down manner. We can observe this in the semantic of `Stack2` given in Figure 2.12. Before going over the semantic, we would like to emphase once again the conciseness of the model we get thanks to priorities. Indeed, without priorities, we would need to keep track of the current state of the stack in order to insert and pop an element in the desired manner.

2.1.3 Examples

We have seen an overview of an AltaRica language. Its modularity is extremely handy and powerfull. Before presenting the semantic of an AltaRica node in more details, we present a few examples that allow us to illustrate different possibilities of the AltaRica language.

Stacks.

We have built for now a two cell stack, this model can easily be extended with more cells. In Figure 2.13(a) we have an AltaRica description of a three cell stack. This node is almost identical to our two cell stack AltaRica node of Figure 2.11(a). There are two differences: the *Stack* subnode is now a `Stack2` AltaRica node, and new synchronization vectors are added. Using a `Stack2` instead of a `Stack1` subnode type for the *Stack* element simply allows us to obtain a larger container. In order to have a proper stack behavior of our new node, we need to adapt and extend the synchronization vectors (in comparison with the `Stack2` node). We first need to adapt the synchronization vectors because the `Stack2` node does not define the same set of events as the `Stack1` node: the `Stack2` node distinguishes insert ($pushT, pushS$), and pop ($popT, popS$) events with respect to their locations, whereas the `Stack1` node does not. As in the `Stack2` node,

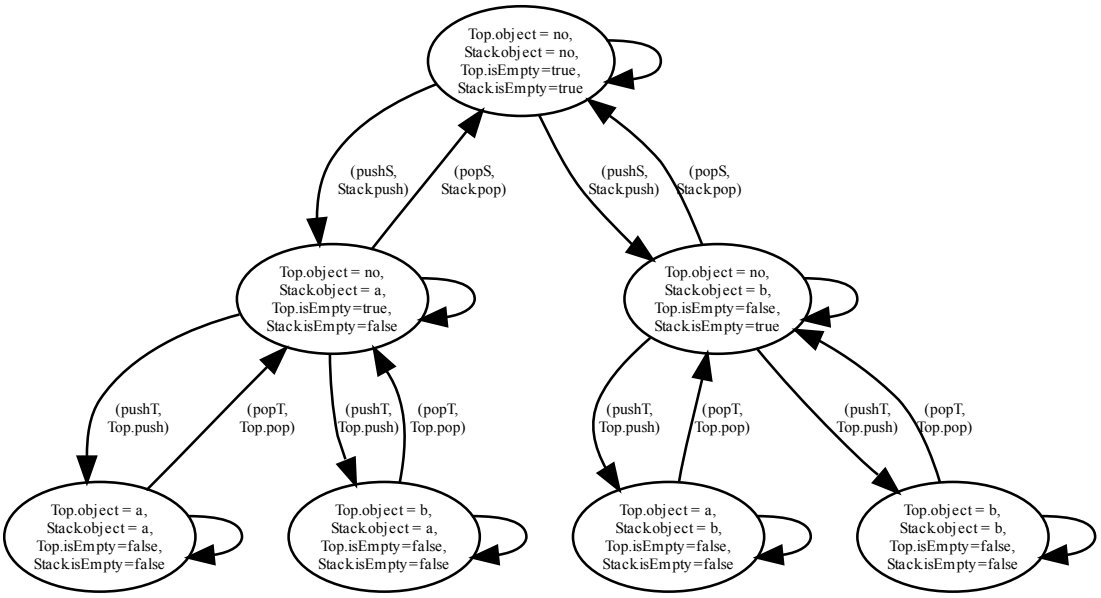


Figure 2.12: The semantic of Stack2.

inserting and popping an element in the top of the stack is simply managed by a direct synchronization of these with the *Top* subnode. Managing the *Stack* related events is slightly more tricky: since the `Stack2` node can insert objects either in its *Top* or *Stack* subnodes (in a stack way) we need to synchronize the *pushS* (resp. *popS*) event of `Stack3` sometimes with the *pushT* (resp. *popT*) and sometimes with the *pushS* (resp. *popS*) of the *Stack* subnode. The subnode *Stack* which is a `Stack2` node guarantees that the object will be inserted and popped in the correct way. It is easily seen that the AltaRica node `Stack3` is a model of a three cell stack.

With our `Stack3` node, we have a model that is now sufficiently generic to allow us to extend it with one or more cells in a parametric way. In Figure 2.13(b) we have a "parametrized" version of our AltaRica model of a stack. Observe that we only need to modify the node name, and the type of the subnode *Stack* to obtain a parametric version of our stack model. Here the parameter is the "variable" N, and for example a `Stack5` node will define a *Stack* subnode which will be a `Stack4` AltaRica node that is defined similarly: its *Stack* subnode is the `Stack3` AltaRica node that we have presented. Note that, the AltaRica language does not at present support parametric models: we need to fix the maximal height of the stack in advance.

FIFO containers.

We present another classical container: the FIFO (First In First Out) container. Compared to stack, we propose here a slightly more dynamic modelization of FIFO container: in our stack model objects where popped "in place", in the model of a FIFO container its

```
node Stack3
  sub
    Top   : Stack1;
    Stack : Stack2;
  event
    pushT < pushS;
    popT > popS;
  trans
    true |- pushT, pushS,
            popT, popS -> ;
  sync
    <pushT, Top.push>;
    <popT, Top.pop>;
    <pushS, Stack.pushT>;
    <pushS, Stack.pushS>;
    <popS, Stack.popT>;
    <popS, Stack.popS>;
edon
```

(a)

```
node StackN
  sub
    Top   : Stack1;
    Stack : StackN-1;
  event
    pushT < pushS;
    popT > popS;
  trans
    true |- pushT, pushS,
            popT, popS -> ;
  sync
    <pushT, Top.push>;
    <popT, Top.pop>;
    <pushS, Stack.pushT>;
    <pushS, Stack.pushS>;
    <popS, Stack.popT>;
    <popS, Stack.popS>;
edon
```

(b)

Figure 2.13: AltaRica models of a stack, (a) An AltaRica model of a three cell stack, (b) An AltaRica model of a N cell stack.

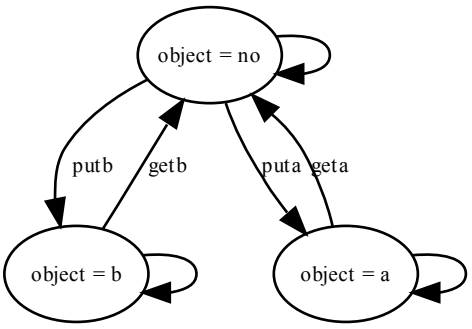
elements "move" toward its head whenever they can. Another difference with our stack model is the basic container that we use, here we will use a modified container that allows us to distinguish (with events) the objects it contains.

In Figure 2.14(a) we have given the AltaRica description of our basic cell container named `Cell`. This node differs from our stack container: the `Stack1` node of Figure 2.4(a). The first difference is the absence of the flow variable *isEmpty* and its associated assertion that kept track of the presence of an object in the container. The second difference is the use of named events: *pusha*, *popa* and *pushb*, *popb* that allow us to distinguish the pushed and popped objects (an object of type a or b) into the cell. The semantic of `Cell` is given in Figure 2.14(b).

Equipped with our `Cell` node, we can now build our FIFO container. We start by a small two cell FIFO, and then see how to extend this container. In Figure 2.15(a) we have the AltaRica description of our `Fifo2` node. This node manipulates two subnodes named *Head* and *Queue* that are `Cell` AltaRica nodes. The *Head* subnode serves as the head of our FIFO, and the *Queue* subnode serves as the remaining of our FIFO. Like the `Cell` node, our `Fifo2` declares a pair of events to insert and get the objects a and b, but it also declares an extra event called $shift$ that will allow us to move objects (from the queue to the head) in the FIFO. The $shift$ event when declared is also given a higher priority than the $put(a/b)$ and $get(a/b)$ events. This ensures that whenever a shift labeled transition can be fired, it will be. A single transition labeled with all the events is defined, this transition allows us to fire any event at any time (at least in the local view of `Fifo2`). The synchronization vectors are elements of our `Fifo2` node.

```
node Cell
  state  object : {no, a, b};
  init   object := no;
  event puta, geta, putb, getb;
  trans
  object = no |- puta -> object := a;
  object = no |- putb -> object := b;
  object = a |- geta -> object := no;
  object = b |- getb -> object := no;
edon
```

(a)



(b)

Figure 2.14: A container cell: (a) the node Cell, (b) the semantic of Cell.

Notice that the get events $get(a/b)$ are synchronized with their related $get(a/b)$ events of the subnode $Head$, and likewise the insert events $put(a/b)$ are synchronized with their related $put(a/b)$ events of the $Queue$ subnode. These synchronizations are quite different from the ones we used in our `Stack3` node: here the head of the container is fixed and is the only location where we can get objects, and we can only insert objects in the queue. In contrast, the `Stack3` model allowed us to insert and get from any location as long as it was done according to a stack behavior. In the `Fifo` node in order to get objects previously inserted in the queue, we need to move them into the head (when the head is empty). To do this, we define the last two synchronization vectors, that synchronize the $shift$ event of `Fifo2` with the $put(a/b)$ of $Head$ and the related $get(a/b)$ of $Queue$. These vectors allow us to get an object from the queue and insert it into the head (one vector is defined for each object a and b). This "move" action must be performed every time that it is possible. This is ensured by the priority given to the shift event.

To illustrate the behavior of `Fifo2` suppose that we want to insert and object a. This action is modeled (and performed) by the transition $\langle puta, \varepsilon, puta \rangle$. This transition inserts an a object into to $Queue$ subnode of `Fifo2`. Now since the $Head$ subnode is empty, thanks to the priority given to the $shift$ event we can only fire the transition labeled $\langle shift, puta, geta \rangle$ that inserts an a object into $Head$ while getting it from $Queue$. Once this is done, we can either get the object a, or insert another object.

Extending this FIFO container with extra cells is straightforward: in order to get a FIFO container of N elements we only need to use an "$N-1$" FIFO container for the $Queue$ subnode. A parametric node that does this modification is given Figure 2.15(b).

2.2 AltaRica Model Semantics

As we have seen, the semantic of an AltaRica node is given as a labeled transition system. Before going over the description of the semantics, we start by a brief presentation of the formal model associated to an AltaRica node.

```
node Fifo2
  sub
    Head  : Cell;
    Queue : Cell;
  event
    {puta, geta} < shift;
    {putb, getb} < shift;
  trans
    true |- puta, putb,
            geta, getb, shift
                -> ;
  sync
    <geta, Head.geta>;
    <getb, Head.getb>;
    <puta, Queue.puta>;
    <putb, Queue.putb>;
    <shift, Head.puta,
       Queue.geta>;
    <shift, Head.putb,
       Queue.getb>;
edon
```

(a)

```
node FifoN
  sub
    Head  : Cell;
    Queue : FifoN-1;
  event
    {puta, geta} < shift;
    {putb, getb} < shift;
  trans
    true |- puta, putb,
            geta, getb, shift
                -> ;
  sync
    <geta, Head.geta>;
    <getb, Head.getb>;
    <puta, Queue.puta>;
    <putb, Queue.putb>;
    <shift, Head.puta,
       Queue.geta>;
    <shift, Head.putb,
       Queue.getb>;
edon
```

(b)

Figure 2.15: An AltaRica model of a FIFO of three cells

Formally, an AltaRica node is a tuple $N = \langle V, \Sigma, G, \preccurlyeq, \delta, I, A, N_0, \ldots, N_n \rangle$, where V is a set of variables (both state variables S and flow variables F), Σ is a set of events, $G = \{(g_i, e_i, u_i)\}$ is a set of guarded transitions, $\preccurlyeq$ is a partial order over Σ that defines a priority relation, δ is a set of synchronization vectors, I is an initial condition, A is an assertion over the variables of V, and $N_0, \ldots, N_n$ are sub AltaRica nodes.

An AltaRica node description define a formal AltaRica node. Some of the elements are explicitly given, others are induced by the description. The state and flow variables of an AltaRica node description define the set of variables V, and their associated domain. Likewise, the declared events together with the ε event define the set Σ.

The assertion in an AltaRica node description is a list of boolean expressions. The conjunction of these expressions defines our formal assertion A (when no assertion is specified, A is set to the value *true*). Recall that, a configuration $\vec{c}$ is a valuation of the variables in V (i.e., an element of $\mathcal{D}(V)$), that satisfies the assertion A. The initial condition I is defined as a set of assignments of the state variables. It is composed of the expressions from init clauses of the AltaRica node description. All configurations $\vec{c}$ that are consistent with I are initial configurations. Note that, if a variable is not assigned in the initial condition, then it is consistent with the initial condition regardless of its value.

The macro transitions declared with multiple events are duplicated in order to get a copy for each event (and the update list is interpreted as a set of assignments like the

initial condition). Recall that the transition whose guard is set to *true*, labeled ε, and does not update the state variables is implicitly declared for all AltaRica nodes. More formally, a guarded transition is a triplet (g, e, u) where g is a guard, e is an event, and u is an update. Given two configurations $\vec{c_1}$, and $\vec{c_2}$ we say that the couple $(\vec{c_1}, \vec{c_2})$ satisfies the guarded transition if the variable valuation of $\vec{c_1}$ satisfies g (denoted $g(\vec{c_1})$) and the configuration $\vec{c_2}$ reflects the assignments of u w.r.t. $\vec{c_1}$ (denoted $\vec{c_2} \in u(\vec{c_1})$). Observe that the update $u(\vec{c_1})$ defines a set of valuations. This is because of flow variables: their values can change to any value that satisfies the assertion. As we have seen in Section 2.1.1 the assignments defined in u can only update state variables. On the other hand, flow variables are constrained by the assertion but otherwise they may change arbitrary with a transition. This generates a set of possible target configurations.

Additionally, the priority relation $\preccurlyeq$ and the synchronization vectors δ are induced by the description. In an AltaRica node description, we only define a priority between events. The priority relation $\preccurlyeq$ is the smallest partial order generated by these pairs of priorities. For instance, when no priority is specified, the partial order $\preccurlyeq$ reduces to the equality relation. The synchronization vectors δ are elements of $\Sigma \times \Sigma_1 \times \ldots \Sigma_n$. Therefore, each synchronization vector of a description must be extended (when necessary) to synchronize an event of the node and an event for each subnode. The ε event will be used as the synchronized event if the node (or a subnode) is not explicitly synchronized in a vector.

Now we can present the semantics of an AltaRica node. First we go over the simplest setting: the semantics of a leaf node. Then we present the semantics of a hierarchical AltaRica node. For hierarchical nodes, we will briefly present two distinct methods: a classical approach that relies on the product of automata. Then we will give a rewriting method that flattens a hierarchical AltaRica node into a leaf node. The equivalence of both approaches is then discussed. Finally, we present a restriction on AltaRica nodes and its impact on the semantic computation.

2.2.1 Semantic of a Leaf Node

The semantic of a leaf node is relatively straightforward. Given an AltaRica node $N = \langle V, \Sigma, G, \preccurlyeq, \delta, I, A \rangle$ the semantic of N is the labeled transition system $S = \langle Q, \Sigma, \rightarrow, I \rangle$ defined as follows:

As said previously, $Q = \{q_c\}$, where $\vec{c}$ is a configuration of the AltaRica node. Put differently, for each configuration $\vec{c}$ their exists a state $q_c \in Q$ that represents it. The set of events of the transition system is exactly the set of events of the AltaRica node. The transition relation is obtained as follows: for each guarded transition (g, e, u) of N if there exists a pair of configurations $(\vec{c_1}, \vec{c_2})$ such that $\vec{c_1}$ satisfies the guard g and $\vec{c_2}$ satisfies the update u w.r.t. $\vec{c_1}$ (i.e., $\vec{c_2} \in u(\vec{c_1})$), then we add the transition (q_{c_1}, e, q_{c_2}) to the transition relation $\rightarrow$ of S. When such a tuple $(\vec{c_1}, \vec{c_2})$ exists, we say that $\vec{c_2}$ satisfies the *post condition* of the transition w.r.t. $\vec{c_1}$. Finally, for each state of S we eliminate all of the outgoing transitions for which there exists another transition labeled with an event of higher priority. This defines the *configuration semantic* of the node. The initial states of S are the states of Q whose configuration satisfy the initial condition of N. The

reachable states, from these initial states, are called in the sequel the *semantic* of the node N.

2.2.2 Semantic of a Hierarchical Node

The first way of computing the semantics for a hierarchical AltaRica node is *semantic composition*. This is a classical method based on automata product. Here we just give an overview of the method, and refer the reader to the theses [Poi00, Vin03] where this method is presented at length.

In this setting, in order to obtain the semantic of a node we proceed as follows: first, we start by computing the semantics of the leafs nodes. Once we have the semantic (i.e., transition systems) of the leaf nodes, we can go one step up in the hierarchy and compute the semantics of their predecessors.

Consider a node N, and assume that we already have the semantics of its subnodes $N_0, \ldots, N_n$. Say that these are transition systems $S_0, \ldots, S_n$. We are going to compute the semantics of N that will be the transition system S. We will need to take into account the following elements:

1. The assertion of the node,
2. the synchronization vectors to determine the events of the semantic,
3. the priorities in order to refine the transition relation, and
4. the initial condition to be able to determine the "reachable" semantic.

States and Assertion

We start with the set of states of S. This set is easily obtained, it suffices to compute Q as the product of the sets $Q_0, \ldots, Q_n$ with the set $\{q_v\}$ where $\vec{v} \in \mathcal{D}(V_N)$ (i.e., a set where each state represent a valuation of the variable of N). Then we eliminate from Q the states whose valuation does not satisfy the assertion of N[3]. Again, we obtain $Q = \{q_c\}$ where $\vec{c}$ is a configuration of N.

Events, Transitions, and Priorities

Unlike a leaf node, the semantics of a hierarchical node does not use the set of events of the AltaRica node directly as its set events. Instead, almost like states, the set of events Σ of S is defined as a particular product as proposed by Arnold and Nivat [AN82].

[3] Recall that the assertion of a node can range over its subnodes variables.

Event & Synchronization vectors The set of events Σ of S (also referred to as *flat* events) is obtained in two steps: We start by extending each synchronization vector with the ε event of each unspecified node. For example the synchronization vector $< geta, Head.geta >$ of the `Fifo2` node in Figure 2.15(a) becomes $< geta, Head.geta, Queue.\varepsilon >$. This vector becomes an event of Σ; it is called flat event since it determines an event in every subnode of the node. Some events of a node (or its subnodes) are not synchronized: they are not part of any synchronization vector. For each of these events e, we create an ε synchronization vector that synchronizes e with the ε event of all other nodes. For example, in Figure 2.15(b) we have a parametric FIFO, suppose that $N = 3$, its *Queue* subnode is a `Fifo2` AltaRica node that defines a $shift$ event. This event is unsynchronized, and therefore a new flat event $< \varepsilon, Head.\varepsilon, Queue.shift >$ will be added to Σ.

The AltaRica language describe a synchrone system, where each node must fire a transition simultaneously. In order to model asynchronous systems, a "no operation" action: the ε event have been introduced. This allow a node to fire a ε transitions and behave asynchronously. This is why the synchronization vectors are extended with ε events of unsynchronized nodes.

Transitions. The transition relation of S is induced by the local variables of the node, its flat event, and its subnodes semantic. Consider $Q_N = \{q_c\}$ where $\vec{c}$ is a configuration of N. The transition relation $\rightarrow_N$ is defined as follows: for each (q_{c_1}, q_{c_2}) of Q_N (two states representing the configurations $(\vec{c_1}, \vec{c_2})$), if there exists a transition (g, e, u) in N such that c_1 satisfies g, and $\vec{c_2} = u(\vec{c_1})$ then we have $q_{c_1} \xrightarrow{e}_N q_{c_2}$. Put differently, the $\rightarrow_N$ transition relation is the transition relation of the semantic of N when stripped off its subnodes and its priority.

Equipped with the $\rightarrow_N$ transition relation, and the set of flat events of N that we denote Σ' we can now define the transition relation of S. Before going into more details, we need to introduce the following notation: given a state q of $\mathcal{D}(V_N) \times Q_0 \times \ldots \times Q_n$, we denote by $q[i]$ the projection of q the projection of q on its i^{th} component. Now consider a flat event $(e, (e_0, \ldots), \ldots, (e_n \ldots))$ of Σ, and every two states q_1, q_2 of Q such that we have $q_1[N] \xrightarrow{e}_N q_2[N]$ and $q_1[i] \xrightarrow{(e_i \ldots)}_{S_i} q_2[i]$ for each subnode N_i of N we add the transition $q_1 \xrightarrow{(e,(e_0,\ldots),\ldots,(e_n\ldots))}_{S_i} q_2$ to $\rightarrow$ the transition relation of S. The transition relation obtained need to be pruned using the priority relation of the node.

Priorities. Applying priorities to eliminate some transitions in S is done as in the leaf node case. The only difference is the labeling of the transitions: in S the transition are labeled with flat events. Since the partial order in N talks only about events of N, only events of N are used to determine the order between flat events. In fact, the order between flat events $(e, (e_0, \ldots), \ldots, (e_n \ldots))$ and $(e', (e'_0, \ldots), \ldots, (e'_n \ldots))$ is determined by the order between e and e' given in N. With this approach we can eliminate transitions according to their priority in the same way as for the leaf case (cf. Section 2.2.1).

Initial Condition

The initial states of S are defined as in leaf nodes: they are the states of S whose configuration satisfy the initial conditions of the node N and of its subnodes. Note that a node can overwrite its subnode initial condition. Recall that, a node (in this case a subnode), can assign a value to its state variables in its initial condition, as we have seen in Section 2.1.1. One or more of these assignments, can be overwritten by its predecessor. To overwrite this assignment the predecessor specifies a new assignment to the states variables of its subnode in its initial condition. As an example the node `Fifo2` of Figure 2.15(a), can with its **init** clause, set its *Queue* subnode state variable *object* to the value a with the following expression: $Queue.object := a$. This would overwrite the original initial condition of the *Queue* `Cell` node: $object := no$.

2.2.3 Syntaxic Flattening

Another approach to define semantics of AltaRica model is *syntaxic flattening*. This method transforms an AltaRica model into one leaf node. This is done with the help of rewriting rules that allow to flatten level by level an AltaRica hierarchy into a single leaf node. This method was proposed by Gérald Point in [Poi00]. As for the composition method above, we here only go over the main points of the method, and refer the reader to [Poi00] for a more detailed presentation.

Leaf Nodes Preparation (Priorities Elimination). Before presenting the flattening of a hierarchical node, we need to prepare the leaf nodes to be lifted into their parent. This preparation will allow us to eliminate the priorities defined in theses nodes.

The idea is to modify the guards of the transition labeled with an event of lower priority. The goal is to be able to fire them only when the transition of higher priority cannot be fired. There are two issues to take into account: the guard and the update of the transitions. In order to fire a transition (g, e, u) from a configuration $\vec{c}$ the guard g must be satisfied in $\vec{c}$ and there must exist another configuration $\vec{c'}$ such that $\vec{c'} \in u(\vec{c})$. In order to determine if such a pair of configurations exits, we use the *post condition* predicate $pc((g, e, u))$ that is *true* if and only if for a configuration $\vec{c}$ that satisfies g their exist $\vec{c'}$ such that $\vec{c'} \in u(\vec{c})$, and $false$ otherwise.

Now, to eliminate priorities we proceed as follows: for each transition, the guard is extended with the conjunction of the negation of the guards of any transition labeled with an event of a higher priority together with the negation of its post condition. For example, given two transitions $t_0 = (g_0, e_0, u_0)$ and $t_1 = (g_1, e_1, u_1)$ such that we have $e_0 \prec e_1$, the transition t_0 is rewritten as: $t'_0 = (g_0 \wedge (\neg g_1 \vee (g_1 \wedge \neg pc((g_1, e, u_1)))), e_0, u_0)$. The new transition t'_0 can be fired only if the transition t_1, whose label has higher priority, cannot be fired. This allows us to manage the priorities defined in the node directly in the guard of the transition.

Consider an AltaRica node N that has a subnode N_0. We will now propose a method that allows us to obtain a *flat* AltaRica node with the same semantics. As expected, we

want to "lift up" the subnode N_0 into the node N. Before, importing the elements of N_0 into N, we need to prepare the subnode N_0. In order to keep track of the subnode's variables, we prefix them with the subnode's name using a dot as a separator in the subnode (yet, when it is clear from the context we simply write the subnode variable name). This renaming is also done for the events of the subnode. With our variables and events renamed, the transitions (guard, events, and updates), and initial condition are rewritten with the new variables and events names.

States, Flows, Initial Condition, and Assertion

Now that our subnode N_0 is ready we proceed as follows. First we import the state and flow variables of N_0 into N. We then replace every reference to the variables of N_0 in N by their imported counterparts. The initial condition of N is then extended with the initial condition of N_0. The new initial condition is the "overwrite union" of the initial conditions of both N and N_0: recall that a node can overwrite its subnode initial condition. We denote this particular union $\uplus$, here for instance $I = I_N \uplus I_0$. Likewise the assertion of N is extended with the assertion of N_0.

Events, Transitions, and Priorities

The set of events is redefined as in the semantic composition method (described in Section 2.2.2) in order to obtain flat events. The synchronization vectors of N are then eliminated since they are already integrated into the flat events.

Transitions. With the flat events, we can now write the new transitions for each flat event using the transitions of N and N_0. Given a flat event $\langle e, e_0 \rangle$, for each transition $t = (g, e, u)$ of N and $t_0 = (g_0, e_0, u_0)$ of N_0 we create a new transition $(g \wedge g_0, \langle e, e_0 \rangle, (u, u_0))$. These transitions once computed replace the transitions of N.

Priorities. To finalize our rewritten AltaRica node N, we rewrite the transition in order to manage the priorities in their guards. This is done as in the case of the leaf node discussed above.

2.2.4 Equivalence of Both Approaches

The semantic composition method and the syntaxic flattening approach are equivalent. The proof is presented in [Poi00]. These two methods still coexist since each approach has its advantage and drawbacks.

The syntactic flattening method is well suited for large hierarchical AltaRica nodes that do not define complex event synchronizations: complex events synchronizations may lead to a large number of flat events (in some cases exponentially larger). The syntactic method allows "on the fly" exploration of the flatten AltaRica node, which cannot be realized with the compositional method.

On the other hand, the composition method can eliminate useless events while composing the nodes, but can suffer from the state space explosion at an intermediate stage of the semantic computation. This can occur even if the final semantics is of a tractable size.

2.2.5 AltaRica without Flow and Assertions

We are now turning our focus towards the computation of the semantics of a subset of the AltaRica language. We consider AltaRica nodes that contain no flow variables, and no assertions. This subset will be the formal model used by our upcoming hierarchical CEGAR algorithm of Chapter 5.

This restriction allows us to simplify the semantic computation of our AltaRica nodes. Without flow variables and assertions, computing the post condition of a AltaRica transition is easier: we do not need to find a valuation of the flow variable that satisfies the assertion and the transition guard. The method we propose is based on a flattening algorithm. The restriction we impose, allows us to modify slightly the original syntactic flattening method.

The flattening algorithm Flatten given in Figure 2.16, is a recursive method applied bottom up on the structure of the AltaRica hierarchy, that returns a leaf AltaRica node N' without priorities semantically equivalent to the hierarchical AltaRica node N.

Leaf Nodes

In the Flatten algorithm the leaf nodes and hierarchical nodes are treated separately. Leaf nodes are treated in the Lines 1 to 11. The algorithm simply eliminates the priorities from these nodes (as presented in the Section 2.2.3). To do so, it modifies the guards of transitions. The modification extends the guard with the formula saying that an event of higher cannot be executed. This allows to preprocess the priorities: for instance consider two transitions $t = (g, e, u)$ and $t' = (g', e', u')$ such that $e \prec e'$, and let (c_1, c_2) be a couple of configurations such that c_1 satisfies both g and g', and c_2 satisfies the post condition of both t and t'. Once the transition (g, e, u) is processed by the Flatten algorithm the transition becomes $(g \wedge (\neg g' \vee (g' \wedge \neg pc((g', e', u'))), e, u)$. Now c_1 does not satisfy the modified guard, and no transition from c_1 to c_2 labeled with e will be present in the semantics. Hence we have eliminated the transitions according to the priorities. Once the transitions are rewritten, the priorities of the node are eliminated and the algorithm returns in Line 11 the new AltaRica node.

Hierarchical Nodes

In the case of an hierarchical AltaRica node, the Flatten (Lines 13 to 27) proceeds differently: it will construct a new AltaRica node out of the node and its flatten subnodes.

In it first steps, Lines 13 through 15, the algorithm flattens the node subnodes, sums the node's and flattened subnode's variables, and computes an initial condition based on the node and on its flattened subnodes initial conditions. This new set of variables

Flatten (N)
Input: N an AltaRica Node.

```
1   if (N is a leaf node) then
2     G' ← ∅
3     for each (g, e, u) ∈ G do
4       g' ← g
5       for each (g'', e'', u'') ∈ G do
6         if (e ≺ e'') then
7           g' ← g' ∧ (¬g'' ∨ (g'' ∧ ¬pc((g'', e'', u''))))
8       done
9       G' ← G' ∪ {(g', e, u)}
10    done
11    return ⟨V, Σ, G', =, ∅, I, true⟩
12  else
13    N'_0, ..., N'_n ← Flatten(N_0), ..., Flatten(N_n)
14    V' ← V ∪ V'_0 ∪ ... ∪ V'_n
15    I' ← I ⊎ (I'_0 ∪ ... ∪ I'_n)
16    Σ' ← FlattenEvents(N, N'_0, ..., N'_n)
17    G' ← ∅
18    for each ((e, (e_0, ...), ..., (e_n, ...)) ∈ Σ' do
19      for each (g, e, u) ∈ G, (g_0, (e_0, ...), u_0) ∈ G'_0, ..., (g_n, (e_n, ...), g_n) ∈ G'_n
20        G' ← G' ∪ {(g ∧ g_0 ∧ ··· ∧ g_n, ((e, (e_0, ...), ..., (e_n, ...)), u ∪ u_0 ∪ ··· ∪ u_n)}
21    done
22    for each (e, e') ∈ (Σ × Σ) such that e ≺ e' do
23      for each ((e, ...), (e', ...)) ∈ (Σ' × Σ') do
24        set ((e, ...) ≺' (e', ...)
25      done
26    done
27  return Flatten(⟨V', Σ', G', ≼', ∅, I', true⟩)
```

Figure 2.16: Flatten algorithm

and initial condition will be part of the flattened version of the input node N. The algorithm then computes the new set of events (flatten events) with the help of the function FlattenEvents given in Figure 2.17.

The FlattenEvents algorithm computes the set of flat events (denoted Σ') of a given AltaRica node N. The algorithm computes this set in two steps: first it computes ε synchronization vectors for unsynchronized events (of the node, then of its subnodes), afterwards it computes the flat events induced by synchronization vectors. The unsynchronized events of the node are treated in the algorithm Lines 3 through 5. For each unsynchronized event an ε synchronization vector is created. This vector synchronizes the event with the ε events of the subnodes. Likewise, the algorithm extends the new set of synchronization vectors δ_X with an ε synchronization vectors for each unsynchronized events of the subnodes Lines 6 through 10. Once the set δ_X extended with these new synchronization vectors we can compute the nodes flat events. The flat events, induced by the new set of synchronization vectors δ_X are treated in the algorithm in the Lines 11

FlattenEvents $(N, N_0, \ldots, N_n)$
Input: N an AltaRica Node, $N_0, \ldots, N_n$ the flatten subnodes of N.

```
1   Σ' ← ∅
2   δ_X ← δ
3   for each e ∈ Σ such that (e, ...) ∉ δ do
4     δ_X ← δ_X ∪ {(e, ε_0, ..., ε_n)}
5   done
6   for each i = 0 to n do
7     for each e_i ∈ Σ_i such that (..., e_i, ...) ∉ δ do
8       δ_X ← δ_X ∪ {(ε, ..., e_i, ..., ε_n)}
9     done
10  done
11  for each (e, e_0, ..., e_n) ∈ δ_X do
12    for each ((e_0, ...) ∈ Σ_0, ..., (e_n, ...) ∈ Σ_n) do
13      Σ' ← Σ' ∪ {(e, (e_0, ...), ..., (e_n, ...))}
14    done
15  done
16  return Σ'
```

Figure 2.17: FlattenEvents algorithm

through 15. Basically, the algorithm will iterate over the set of synchronization vectors, and add for each synchronization vector one or more new flat events into the set Σ'. For a given synchronization vector $(e, e_0, \ldots, e_n)$ of δ_n, the algorithm will create a new flat event of N using the flat events of its subnodes $N_0, \ldots, N_n$ that are of the form $(e_i, \ldots)$.

Once the flat events are computed, the Flatten algorithm computes a new set of transitions using the flat events. In more details, for each flat event $((e, (e_0, \ldots), \ldots, (e_n, \ldots))$ the algorithm identifies the transitions of N labeled with the event e, and identifies the transitions of the flatten subnodes $N_0, \ldots, N_n$ labeled with $(e_0, \ldots), \ldots, (e_n, \ldots)$ and creates a new transition combining the guards and the updates into a new transitions[4]. A new priority relation $\preccurlyeq'$ is also computed. This new priority relation $\preccurlyeq'$ allows us to keep track of the priority defined over the events of the node on the flatten events. Finally in order to eliminate the priorities of the node the Flatten is called with the new sets computed and the new priority relation: $\langle V', \Sigma', G', \preccurlyeq', \emptyset, I', true, \emptyset \rangle$ which put together form an AltaRica leaf node.

Example

To illustrate the Flatten algorithm, consider `Fifo2` node of Figure 2.15(a) and let us apply the Flatten algorithm to it. When invoked with the AltaRica node `Fifo2`, the algorithm starts by determining if `Fifo2` is a leaf node (Line 1). This node defines two subnodes therefore the algorithm jumps to the Line 11. Then it flattens the two subnodes *Head* and *Queue* by calling Flatten(*Head*) and Flatten(*Queue*). These subnodes are both `Cell` AltaRica nodes (see Figure 2.14(a)). The `Cell` node is a leaf node that do not define a

[4]If no transition exist labeled with the given (flatten) event then, the tuple is discarded.

priority relation over its events. Therefore, the execution of both Flatten($Head$) and Flatten($Queue$) returns the nodes unchanged. Then the new set of variables is computed[5] and is $V' = \{Head.object, Queue.object\}$, and the initial condition is $I' \equiv Head.object = no \wedge Queue.object = no$. In its next step the algorithm calls the FlattenEvents algorithm with `Fifo2`, $Head$, and $Queue$ as its arguments. FlattenEvents will return flatten events of `Fifo2` which is exactly the flat events induced by the synchronization vector of `Fifo2` together with the event: $(\varepsilon, (Head.\varepsilon), (Queue.\varepsilon))$. For example the synchronization vector $\langle geta, Head.geta\rangle$ generates the flat event: $(geta, (Head.geta), (Queue.\varepsilon))$. The algorithm continues and computes the new set of transitions G'. For the `Fifo2` node, it will generate the following transitions:

- $(Head.object = a, (geta, (Head.geta), (Queue.\varepsilon)), Head.object = no)$
- $(Head.object = b, (getb, (Head.getb), (Queue.\varepsilon)), Head.object = no)$
- $(Queue.object = no, (puta, (Head.\varepsilon), (Queue.puta)), Queue.object = a)$
- $(Queue.object = no, (putb, (Head.\varepsilon), (Queue.putb)), Queue.object = b)$
- $(Head.object = no \wedge Queue.object = a, (shift, (Head.puta), (Queue.geta)), Head.object := a,\ Queue.object := no)$
- $(Head.object = no, Queue.object = b, (shift, (Head.putb), (Queue.getb)), Head.object := b,\ Queue.object := no)$
- $(true, (\varepsilon, (Head.\varepsilon), (Queue.\varepsilon)), \emptyset)$

Now that the new transition relation is computed, the algorithm goes on and defines the $\preccurlyeq'$ partial order. This is done from Line 22 to Line 26. The priority $\preccurlyeq'$ is defined using the nodes priority $\preccurlyeq$. The algorithm identifies all comparable pairs of events e and e' of Σ such that $e \prec e'$. Such pairs of events are used to define the new order $\preccurlyeq'$ on newly constructed flat events by putting $(e, \ldots) \preccurlyeq', (e', \ldots)$. For the example of the `Fifo2` node, the algorithm will generate for the flat event $(shift, (Head.puta), (Queue.geta))$ the following priorities:

- $(geta, (Head.geta), (Queue.\varepsilon)) \prec' (shift, (Head.puta), (Queue.geta))$
- $(puta, (Head.\varepsilon), (Queue.puta))) \prec' (shift, (Head.puta), (Queue.geta))$
- $(getb, (Head.getb), (Queue.\varepsilon)) \prec' (shift, (Head.puta), (Queue.geta))$
- $(putb, (Head.\varepsilon), (Queue.putb))) \prec' (shift, (Head.puta), (Queue.geta))$

Similar priorities will be defined for the $(shift, (Head.puta), (Queue.geta))$ event. Note that, as usual, the $\preccurlyeq'$ relation is the is the smallest partial order generated by the listed pairs of events.

[5] Observe that the subnodes variables are prefixed with there respective subnode id.

The algorithm has at this point computed all of the elements need to eliminate the subnodes. The final recursive call is done Line 27 to eliminate the priorities from the new node. As previously discussed, the Flatten algorithm eliminates the priorities of leaf nodes using a rewriting method that modifies the guards of events of low priority. For our `Fifo2` node the transition labeled $(putb, (Head.\varepsilon), (Queue.putb))$ will have its guard updated to the following expression: $Queue.object = no \wedge \neg(Head.object = no \wedge Queue.object = a) \wedge \neg(Head.object = no \wedge Queue.object = b)$. This does not change the transition guard since the expression can be simplified to the original guard. In this example, the negation of the post condition does not modify the guard, and for the sake of clarity we left out the expression produced by this negation. However, consider a Fifo system of more than two cells `Fifo3` for example. In this case the rewriting will produce a guard that forbids the insertion of a new object into the cell until any object present in the middle cell (the $Queue.Head.object$ variable) have been shifted to the head to the Fifo (the $Head.object$ variable). Going back to our `Fifo2` example, once the algorithm terminates, the leaf AltaRica node obtained is an priority free AltaRica node semantically equivalent to the `Fifo2` node.

Chapter 3

State of the Art

3.1 Model Checking

Formal verification appeared as an answer to the growing demand of safety in the design of critical systems. Critical systems are more and more present in our today's society. By critical system, we mean any system whose failure can jeopardize human life, or economic liability. In the recent history, many examples of critical systems failures have caused human loss, as well as economical threats to businesses. One of the most compelling examples is the Ariane 5 launch that due to a register overflow forced the operating center to destroy the space shuttle a few seconds after takeoff. Examples of buggy design can be found in various industries from medical equipment (e.g. Therac-25) to the microchip industry (e.g. Pentium Pro and Pentium II FPU bug). To tackle this issue, model checking [CGP99, Cla08, BK08, RCB, WLBF09] has been proposed to improve the confidence one can have in a system by introducing formal verification in the production process of critical systems.

3.1.1 Explicit Model Checking

Explicit state model checking is a first example of formal verification methodology. In the simplest variant, the goal is to verify that a given set of faulty behaviors cannot occur by testing all possible behaviors of the system. To this end, the system is usually represented as a transition system and the task is to explore it an efficient way taking into account the property to be verified. Many tools performing explicit state model checking have been implemented. Two examples are Spin [Hol97], and ARC [Poi00, Arc10]. The first is one the most popular model checkers, the second is the standard explicit and symbolic model checker for AltaRica.

State Space Explosion

Explicit-state model checking, is a verification method where the system states are represented individually: each state and transition is represented as a separate object. This permits to use efficient graph exploration algorithms to check the existence of a counterexample.

The main issue to handle when dealing with explicit-state model checking is computing and maintaining a concrete representation of the state space and of the transition relation of the overall system. In many languages such as Promela, NuSMV, and AltaRica (see Chapter 2) the models are described by modules/components and their interactions. Even if a given module taken separately has a modest size, the overall system has a size that is exponential in the number of modules. Industrial models are usually made of a many small modules. Their number often renders explicit-state model checking impossible.

An approach to alleviate this state-explosion problem is to represent the state space in a compressed form. This approach is commonly called "Symbolic Model Checking". The remainder of this section presents two symbolic approaches to model checking.

3.1.2 Symbolic Model Checking

Symbolic model checking [BCM+92] intends to tackle the state space representation issues by different approaches. One technique is the use of a compact representation of the set of states. For examples, binary decision diagrams (BDD for short) are often used to represent sets of states, and even transition relations. With the use of BDDs it is possible to manipulate a system of more than 10^{20} states. Another approach to the model checking problem is the symbolic exploration of the model. In this type of setting, instead of keeping track of the model's states, the model is transformed into a SAT problem [BCC+03]. This permits the use of a SAT solver to symbolically explore the state space of the model.

BDDs

Binary Decision Diagrams [Bry86] are concise representations of boolean formulas. Practically, a BDD is a rooted directed acyclic graph where each non-terminal state represents a variable, and the two terminal states represent the truth values $true$, and $false$. BDDs can be reduced and ordered (ROBDD for short). A ROBDDs is a canonical representation of sets of values. More importantly, there exists efficient algorithms for ROBDDs composition (union, intersection, negation, ...) see [Bry86, And97]. In the following, by a slight abuse of notation we use BDD for ROBDD

In the context of model checking, BDDs are commonly used to represent a set of states as well as the transition relation of a model. Doing so, it is possible to compute the set of reachable states (w.r.t. some initial states), and determine if these states satisfy a given property. Thanks to BDDs, it is possible to keep track of the reachable states, in a concise way.

Formula Based Approachs & Bounded Model Checking

The formula based approach tackles state space exploration issue differently. Instead of keeping track of the reachable states, the intent here is to explore the state space symbolically. Since the goal of a model checking method is to determine if a model satisfies a property or not, it suffices to determine the existence of a counterexample. This decision problem can be reduced to (and implemented as) a satisfiability problem of a propositional formula. Biere et al. present a survey of these methods in [BCC+03].

These methods gained in popularity as the performance of SAT solvers grew. Yet, the use of a formula to explore the state space induced a completeness issue: A formula symbolically explores the system until a certain depth (referred to as a threshold). An arbitrary threshold k cannot guarantee completeness: a counterexample of length $k+1$ may exist. Determining a completeness threshold is however possible, and discussed in [BCC+03], but in the worst case it can be as large as the number of states of the system.

An induction method proposed by Sheeran et al. [SSS00] also relies on the formula transformations and SAT solvers. Intuitively, the idea is to prove with the help of a SAT solver, that the property is satisfied in initial states, and if a state satisfies the property, then all its successors also satisfy the property. In other words, to show that the property is an inductive invariant. This method guarantees completeness, but may require strengthening inductive invariants by the user, when the inductive step is not verified.

The use of abstractions, in model checking, has then emerged to tackle the state-space explosion issue when BDD and formula based approaches reached their limits. This approach takes its root in the abstract interpretation scheme proposed by Cousot [CC77] in 1977. This work set the foundation of an abstraction-based approach, as system (programs,...) where not directly interpreted, but instead "projected" against the property under verification. The next section of this chapter, is dedicated to the presentation of a prominent abstraction based verification method: CEGAR.

3.2 CEGAR

CEGAR. **C**ounter**E**xample **G**uided **A**bstraction **R**efinement is a successful method for verification of safety properties. It is based on automatic refinement of abstractions of the system under verification (e.g., [CGJ+03, HJMS02, SG04]). It means in particular that construction of the whole state space is not needed. Thus, a priory, the method can help to avoid the state explosion problem, and can be applied to infinite state systems.

The CEGAR algorithm for verification of a safety property is a loop that can be described as follows (see Figure 3.2 page 42). At each iteration of the loop there is some abstraction of the model. If this abstraction verifies the property then the loop terminates returning "model safe". Otherwise a counterexample is found in the abstraction, and this counterexample is executed in the model. If the counterexample is feasible, the loop terminates returning "model unsafe". If it is not feasible then the abstraction is refined

to eliminate the counterexample and the loop is restarted. The refinement step is a subtle point of the algorithm, and depends very much on the abstraction that is used.

3.2.1 Abstraction

An abstraction of a transition system is another transition system that has all the behaviors of the original one.

Given two transition systems S_c and S_a, a *simulation relation* [Mil71] ρ from S_c to S_a is any relation $\rho \subseteq (Q_c \times Q_a)$ satisfying: for all $(q_{c_0}, q_{a_0}) \in \rho$ and $q_{c_0} \rightarrow_c q_{c_1}$ there is a state q_{a_1} s.t. $q_{a_0} \rightarrow_a q_{a_1}$ and $(q_{c_1}, q_{a_1}) \in \rho$. Using a simulation relation between two transition systems, an abstraction relation is defined as follows:

Definition 3.1. *Given two transition systems S_c and S_a, we say that S'_a is an* abstraction *of S_c ($S_c \preceq S_a$), iff there is a simulation relation ρ from S_c to S_a, and the following two properties hold:*

- *For every state q_c of I_c, there exists a state q_a in I_a such as $(q_c, q_a) \in \rho$.*
- *For every (q_c, q_a) in ρ: if q_c belongs to F then q_a belongs to F_a.*

Observe that it is possible to have finite abstractions of infinite transition systems. Thus, one can hope to model check properties of systems that explicit model checking fails to manipulate. An abstraction $\widehat{S}$ of a transition system S satisfies, by definition, $Path(S) \subseteq Path(\widehat{S})$, and consequently, it also satisfies $Run(S) \subseteq Run(\widehat{S})$ (see Definition 1.2). Therefore, abstractions are suitable (and practical) for verification of safety properties.

Boolean Predicate Abstraction

A framework for the automatic generation of abstractions using a set of predicates has been proposed by Graf and Saidi [GS97]. In their framework a system consists of a set of processes where each process has a set of variables, a set of guarded transitions, and an initial condition. The global system is the parallel composition of the set of processes. Its set of variables (resp. guarded transitions) is the union of the processes variables (resp. guarded transitions), and the initial condition is the conjunction of all initial conditions.

The semantics of the global system is a transition system, where the set of states represents the valuations of each variable. The initial states are those who satisfy the initial condition. The transitions are induced by the guarded transitions.

For a given transition system S, an abstract transition system $\widehat{S}$ is defined using a set of predicates $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ on S. The set of states $\widehat{Q}$ of $\widehat{S}$ is the set of subsets of Φ. We can then define

- $\alpha(q) = \{\varphi_i \in \Phi \mid \varphi_i(q)\}$,
- $\gamma(\widehat{q}) = \{q \in Q \mid \alpha(q) = \widehat{q}\}$.

This way $(\alpha : \mathcal{P}(Q) \to \widehat{Q}, \gamma : \widehat{Q} \to \mathcal{P}(Q))$ is a Galois connection, and it satisfies the following properties:

- $I \subseteq \gamma(\widehat{I})$,
- If $q \to q'$ then $\widehat{q} \to \widehat{q}'$.

The abstract system is $\widehat{S} = \langle \widehat{Q}, \to, \widehat{I} \rangle$. The abstract system exhibits all behaviors of the original system, but may introduce some new behaviors too. Note that the transitions of the abstract system are called *may* transitions, because a transition between two abstract states exists if there are two representatives in the respective states for which the transition exists. One can also consider must transitions that we will present in the sequel of this section.

Cartesian Predicate Abstraction Cartesian predicate abstraction is another method to obtain an abstraction of a model. Also introduced in [GS97], cartesian abstractions further abstract boolean abstraction with a 3-valued logic whose values are `true`, `false`, and $*$ for "don't care". This permits generation of smaller and coarser abstractions since the state space is no longer partitioned. This particular type of abstractions have been used for the verification of C programs in the SLAM project [BPR03, BR01].

3.2.2 Verification of Abstract Counterexamples

As we have noted in the previous section, the non-existence of an abstract counterexample implies the safety of the abstracted model. In contrast, existence of an abstract counterexample is not conclusive, and requires further analysis. An abstract counterexample is a symbolic representation of a sequence of concrete states and events. The analysis of an abstract counterexample has to determine if the counterexample is feasible or spurious. We say that a counterexample is *feasible* if it induces a path in the concrete model, and otherwise the counterexample is called *spurious*.

In [CGJ+03], Clarke et al. propose a straightforward method based on the computation of the iterated concrete *post* operator over the abstract states and transitions that form the abstract counterexample. For the simplicity of presentation we assume that states of the abstract system are sets of states of the concrete system. The proposed algorithm is given in Figure 3.1. Intuitively, at each step, the algorithm determines which concrete states from the abstract state are indeed reachable by the path under consideration. For example, after the first iteration of the loop, the set X of reachable states is equal to $post(\widehat{q}_0 \cap I) \cap \widehat{q}_1$. In the next iteration X is $post(post(\widehat{q}_0 \cap I) \cap \widehat{q}_1) \cap \widehat{q}_2$, and so on. If at some point X is empty, the algorithm terminates and returns "spurious counterexample". Otherwise, the algorithm returns "feasible counterexample".

The algorithm can be viewed as a bounded and restricted model checking procedure: The concrete state space is explored for defined threshold, and the exploration is restricted to the concrete states abstracted by the abstract counterexample.

VerifyPath $(S, \widehat{\pi})$
Input: A transition system S, an abstract counterexample $\widehat{\pi}$.

```
1  X = q̂_0 ∩ I
2  i = 0
3  while i < |π| ∧ X ≠ ∅ do
4      i = i + 1
5      X = post(X) ∩ q̂_i
6  done
7  if X ≠ ∅
8      return " feasible  counterexample"
9  else
10     return "spurious counterexample"
```

Figure 3.1: The algorithm VerifyPath.

Another classical approach to abstract counterexample verification is based on SAT/SMT solvers. In this case, the sequence of abstract states and transitions is written as a formula (using the states predicates and/or the variable updates labeling the abstract transitions). This later approach have been successfully implemented in popular tools such as BLAST [HJMS02] and SLAM [BR01].

3.2.3 Abstraction Refinement

When a spurious counterexample is identified in an abstraction, it is necessary to eliminate it in order to go on with the verification process. The elimination step is referred to as "abstraction refinement".

Recall that a spurious counterexample is an abstract path that does not represent a concrete one. Yet, some prefixes of the abstract counterexample remain "feasible". This means that until some point it is possible to find a concrete path that induced the abstract one. The usual method to refine an abstraction is to identify the longest feasible prefix, and refine its last state (referred to as the failure state).

In a spurious counterexample, there always exists a failure state, since otherwise the counterexample would be feasible. The VerifyPath of Figure 3.1 presented in Section 3.2.2 permits the identification of the longest feasible prefix, and more importantly the identification of the failure state. The *failure state* of a spurious abstract counterexample, is the first state $\widehat{q}_i$ along the counterexample such that the algorithm VerifyPath$(S, \widehat{q}_0 \dots \widehat{q}_i)$ returns "feasible", and VerifyPath$(S, \widehat{q}_0 \dots \widehat{q}_{i+1})$ returns "spurious". Note that the algorithm VerifyPath can be easily modified to return the failure state: It suffices to modify the algorithm so it returns the last non-empty set of concrete states X together with its index $i-1$.

In [CGJ+03] Clarke et al. proposed a partition of the set of concrete states abstracted by a failure state. They identify three types of concrete states: deadend states, bad states, and irrelevant states. These sets of concrete states are defined as follows:

- The *deadend states*: states that are reachable from concrete initial states along the path of the abstract counterexample. These states do not have outgoing transitions to the states abstracted by the next state of the counterexample.

- The *bad states*: states that have an outgoing transition to the next state of the counterexample.

- The *irrelevant states*: every state that is neither a deadend nor a bad state.

To eliminate the spurious counterexample, it suffices to abstract the deadend and bad states by two distinct abstract states. This will eliminate the abstract counterexample. Yet, finding the coarsest refinement that separates the deadend states from the bad states is NP-hard (the proof is given in [CGJ+03]). To tackle this problem, various refinement heuristics have been proposed.

Two "direct" methods have been proposed by Clark et al. in [CGJ+03] and Shoham et al. in [SG04]. We say that these methods are direct because they split the failure state so that the deadend states (resp. bad states) are abstracted by one abstract state, and the remaining concrete states are abstracted together by another abstract state. These two refinement heuristics are implemented in our CEGAR tool, and will be discussed in more details later.

Another approach to this set separation problem has been proposed by McMillan [McM04]. The method relies on Craig interpolants [Cra57]: given two formulas f_1 and f_2 such that $f_1 \wedge f_2$ is unsatisfiable, an interpolant for (f_1, f_2) is a formula f expressed over the common variables of f_1 and f_2 such that $f_1 \Rightarrow f$ and $f \wedge f_2$ is unsatisfiable. For refinement purpose, McMillan proposed in [McM04] an interpolant based predicate generation method that ensures the separation of deadend states from bad states.

3.2.4 The CEGAR Verification Method

Equipped with our abstraction methods, abstract counterexample verification method, and refinement heuristics methods we can now present the CEGAR scheme in more detail. In Figure 3.2 we have represented the CEGAR scheme. On the left hand side we have the system to analyze S and a reachability property φ. The system S is abstracted by an abstraction method that can be one of the methods presented in Section 3.2.1. The model-checker is run on the abstracted system $\widehat{S}$. The only thing required from the model-checker is that it can decide reachability properties and return a witness path if it exists. Depending on the answer from the model-checker the loop can either stop and decide that $S \models \varphi$, or analyze the abstract counter-example. In Section 3.2.2 we have presented a classical abstract path verification algorithm that can be used to verify the feasibility of the abstract path $\widehat{\pi}$ given by the model checker. If the path is feasible, then $S \not\models \varphi$, otherwise the abstraction is refined using a refinement heuristic as those presented in Section 3.2.3.

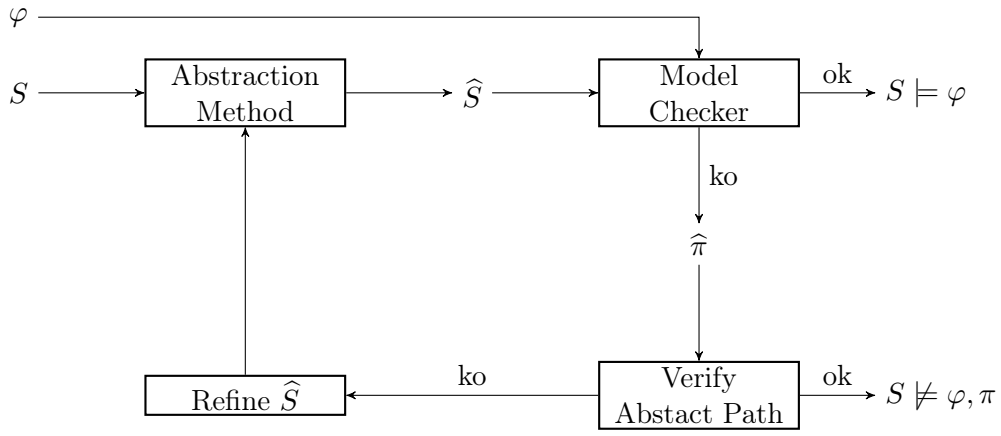


Figure 3.2: A schematic representation of the CEGAR method.

Lazy Abstraction: an Implementation of the CEGAR Loop

Lazy Abstraction [HJMS02, McM06] is a particular implementation of the CEGAR loop. In this setting, instead of generating a complete abstraction of the system (usually a C program), the abstraction is generated while exploring the program structure. In more details, a depth first search is performed on the program structure to determine some possible counterexample. The example is analyzed and if proved spurious invariants will be generated to label the control locations of the program. This will be the first abstraction. As the process continues, new counterexamples will be analyzed and used to update this "on the fly" generated abstraction.

3.2.5 Improvements of the basic techniques

The original CEGAR loop can be accelerated and improved using different methods. We now describe some of these methods.

Modal Transition Systems As Abstraction

Modal transition systems [LT88, Lar89, LX90, LSC95] have been proposed as an extension of Kripke structures since the 80's. Modal transition systems extend classical transition systems with a set of "Must" transitions (classically denoted $\xrightarrow{+}$) that were intended to enforce a given behavior at various points (in the context of process algebra when dealing with specifications).

When dealing with abstractions, a must transition is a transition between two abstract states such that every concrete state from the source abstract state has a concrete successor in the target abstract state. In [SG04] Shoham and Grumberg proposed a CEGAR method that takes advantage of these transitions. In particular they where able to extend CEGAR to checking CTL properties.

In [BKY05, BKS07], Ball et al. proposed to extend modal transition systems with another set of must transitions (denoted $\xrightarrow{-}$) that are the dual of the $\xrightarrow{+}$ must transitions. With the help of these two sets of must transitions they defined the notion of "weak reachability" which entails the existence of a concrete path from a particular sequence of must transitions.

In Chapter 4 Section 4.2, we will present another method that can also take advantage of these must transitions.

Abstraction Slicing

In [BDFW08], Brückner et al. formalized and extended some possible reductions that can be performed on an abstraction that are sound with respect to a given safety property.

The first slicing process is abstract states elimination. In [BDFW08], two types of abstract states are eliminated: "Inconsistent Nodes" that are abstract states that do not represent at least one concrete state, and "Unreachable Nodes" that are abstract states that are not reachable nor coreachable from initial and the error abstract states.

Another classical slicing approach is also exploited in [BDFW08]: live variables pruning. In their setting, transitions are guarded with expressions (as AltaRica nodes see Chapter 2 Section 2.1.1). With the help of live variables computation, the guards of the transitions are simplified: each clause defined over non live variables is eliminated. Another abstraction simplification method proposed is "Bypass Transitions". New transitions are added between two states to bypass an intermediate state that separates them. The corresponding incoming and outgoing transitions of the intermediate state are eliminated (this helps abstract states elimination). To ensure soundness, the bypass transition added reflects the guards of the eliminated transitions.

Path Slicing.

Path slicing [JM05] is yet another approach to the verification of an abstract counterexample. It is a static analysis method that prunes away irrelevant parts of a program path (here viewed as an abstract counterexample). The analysis is done backward from the final state (error location) to the initial state (program entry point). The goal is to determine a set of "live" variables, whose values determine whether or not the abstract counterexample is feasible. Starting at the error location with the variables characterizing it (the live variables at this point), the process goes one step back to the previous location. Then it determines if the operation performed at this location has an impact (modifies) the live variables. If it does, the variables implicated in the operation are added to the live variables set, otherwise they are not added. The process continues until it reaches the initial state of the path. Once these live variables are computed, the abstract path is separated into sub paths whose feasibility will determine the feasibility of the abstract path. In more details, any subsequence of the path that does not modify a live variable is pruned away. This pruning method minimizes the size of counterexample and simplifies analysis of its feasibility.

3.2.6 CEGAR Model Checkers

BLAST. BLAST [BHJM07] (Berkeley Lazy Abstraction Software Verification Tool), is a model checker for C programs that verifies safety properties (reachability of label in a C program). BLAST is also a "mix" model checker as it uses both BDDs and theorem provers to implement a CEGAR loop. A particularity of BLAST is the use of Lazy Abstractions [HJMS02]: the abstraction is constructed "on demand" as the program is analyzed. In fact, the abstraction is an abstract reachability tree, where each node of the tree is labeled with a program location, a list of predicates, and a boolean formula over the predicates. The list of predicates is extended to eliminate as needed spurious counterexamples. The BDDs are used to represent the concrete states abstracted by the nodes (the boolean formula of the node), and two theorem provers are used: one to compute the abstract *post* of a node, and another (interpolating) one to generate new predicates to refine the abstraction.

Yasm. Yasm [GWC06] (Yet Another Software Model-checker), is a CTL model checker for C programs. One of the advantages of Yasm is that it can "prove and disprove properties with equal effectiveness". Yasm is a "mix" model checker as it uses both BDDs and theorem provers to implement a CEGAR loop. The abstraction is a Mixed Transition System [GC06], that combines may and must transitions. A BDD library is used to encode the transition relations, and the theorem prover is used to mine new predicates that will refine the abstraction when a spurious counterexample is discovered.

3.3 Compositional Model Checking

The modelization of a system is often given by a set components, and specification of interactions between components. Compositional model checking exploits this component based description in order to prove or disprove a property of the model. This approach relies on analysis of the individual constituents of a model in order to decide if the model satisfies a given property. In other words, the issue here is to deduce if a given model $M = M_1 \parallel M_2 \parallel \ldots \parallel M_n$ satisfies a property φ from properties of the components $M_1, \ldots, M_n$.

3.3.1 Compositional Reachability Analysis

A CEGAR apprach to compositional reachability analysis have been proposed by Chacki et al. in [COYC03]. In this work, the goal is to perform safety verification on concurent C programs. The problem is presented as a language theoretic property: $\mathcal{L}(C_1 \parallel C_2 \parallel \ldots \parallel C_n) \subseteq \mathcal{L}(\varphi)$ where $C_1, \ldots, C_n$ are programs, and φ is a safety property. In otherwords, we intend to verify automatically that the behaviours of a system composed of concurent C programs remains within a set of "allowed" behaviours.

The framework proposed relies on two orthogonal levels of abstraction namely predicate abstraction, and action-guided abstraction. The verification is performed on the abstractions: instead of computing the product automata of the C programs (represented

as transition systems), a product automata for the two level of abstraction is used. Since the abstractions have a significantly smaller state space, the product automata can be computed. Yet the abstraction introduces spurious behaviors that need to be eliminated. To this end, some abstractions are identified and refined when a spurious counterexample is exhibited.

The programs $C_1, \ldots, C_n$ are each abstracted, as transition systems, using predicate abstraction [GS97] (see Section 3.2.1). The sets of predicates used to abstraction each program are denoted $\mathbb{P}_1, \ldots, \mathbb{P}_n$. Classically, in these abstractions an abstract state is a representation of a set of concrete states, and a existentiel transition relation is computed over the abstract states. As refinement is performed to eliminate spurious behaviors of some abstraction, the number of predicates may grow and lead to abstraction with a large state space: exponential in the number of predicates. To further reduce the abstract state space, an action-guided abstraction of each abstraction induced by the sets of predicates $\mathbb{P}_1, \ldots, \mathbb{P}_n$ is computed. We denote $\widehat{C}_1, \ldots, \widehat{C}_n$ the predicate abstractions of $C_1, \ldots, C_n$ induced by the sets of predicates $\mathbb{P}_1, \ldots, \mathbb{P}_n$.

Given a transition system S, an *action-guided abstraction* of S is a partition based abstraction A. The partition is induced by an equivalence relation over the set of outgoing actions, such that two states are equivalent if and only if they share the same set of ougoing events. In [COYC03] an action-guided abstraction A_i of each predicate abstraction $\widehat{C}_i$ is computed. The initial action-guided abstraction can at most carry as many abstract states as its predicate counterpart. Yet this only applies to the initial action-guided abstraction, when it is refinemed its state space may grow larger than its predicate counterpart.

The two-level CEGAR algorithm starts with a set of C programs $C_1, \ldots, C_n$, and a safety property φ. First it computes a set of predicates abstractions using the sets $\mathbb{P}_1, \ldots, \mathbb{P}_n$ for each $C_1, \ldots, C_n$. Then, an action-guided abstraction $A_1, \ldots, A_n$ of each predicate abstraction $\widehat{C}_1, \ldots, \widehat{C}_n$ is computed. If no counterexample for φ is found in $A_1 \parallel \ldots \parallel A_n$ then the algorithm terminates and returns *true* to the user. Otherwise a counterexample is generated, and analyzed as follows: If the counterexample is not a behavior of $\widehat{C}_1 \parallel \ldots \parallel \widehat{C}_n$ then the action-guided abstractions $A_1, \ldots, A_n$ are refined in order to remove this spurious behavior. Otherwise, if the counterexample is a behavior of $C_1 \parallel \ldots \parallel C_n$ it is returned to the user. If the counterexample is found not feasible the predicate abstraction $\widehat{C}_1, \ldots, \widehat{C}_n$ are refined and the action-guided abstractions are adjusted in consequence. Then the process starts over. So, this algorithm can be seen as a two level abstraction/refinement algorithm.

This approach have been implemented in a tool called MAGIC, experiemental results of the tools are given in [COYC03].

3.3.2 Assume-Guarantee Methods

In 1985, Pnueli [Pnu85] proposed and advocated the use of compositional reasoning in model checking. In this work, he proposed the now well known, and widely used "Assume-Guarantee" paradigm for the verification of safety properties. This assume-guarantee

paradigm is given as this inference rule:

$$\frac{\begin{array}{ll}\text{(Step 1)} & \langle A\rangle M_1\langle\varphi\rangle \\ \text{(Step 2)} & \langle true\rangle X\langle A\rangle\end{array}}{\langle true\rangle\ M_1 \parallel X\ \langle\varphi\rangle}$$

This rule can be rephrased by: if there exists an assumption A under which M_1 satisfies φ, and such that X satisfies A, then the composition of M_1 and X satisfies φ. The component X is viewed as the "environment" of M_1: usually it is the composition of all of the model components but M_1. The verification of the second precondition $\langle true\rangle X\langle A\rangle$ can be performed by reapplying the rule recursively. Let us note by $X_i = M_i \parallel \ldots \parallel M_n$, and $A_0 = \varphi$ we get:

$$\frac{\begin{array}{ll}\text{(Step 1)} & \langle A_i\rangle M_i\langle A_{i-1}\rangle \\ \text{(Step 2)} & \langle true\rangle X_{i+1}\langle A_i\rangle\end{array}}{\langle true\rangle\ M_1 \parallel X\ \langle\varphi\rangle}$$

The main challenge in this approach is an automatic generation of the assumptions $A_1, \ldots, A_n$. To this end, we will present a framework that automatically generates the assumptions. This framework uses a learning algorithm (L^*).

3.3.3 Learning based methods

Automatic learning has been proposed to generate proof assumptions. The methods presented below are based on the L^* algorithm that learns a finite automaton from the series of examples and tests.

The L^* Algorithm

The L^* algorithm [Ang87, RS89] builds a minimal deterministic finite automaton recognizing a particular regular language $\mathcal{U}$ (over an alphabet Σ fixed in advance) from queries and counterexamples. The algorithm acts as a "Learner", and requires a "Minimal Adequate Teacher" who knows the language $\mathcal{U}$ that the learner has to learn. The teacher can answer two questions that the learner can ask:

Membership queries Does a word $w \in \Sigma^*$ belong to $\mathcal{U}$?

Conjectures Does the automaton $\mathcal{C}$ recognize the language $\mathcal{U}$? If $\mathcal{L}(\mathcal{C}) \neq \mathcal{U}$ the oracle returns a counterexample in the symmetric difference of $\mathcal{U}$ and $\mathcal{L}(\mathcal{C})$.

The learner in the L^* algorithm builds a set of prefixes S, and suffixes E over Σ, who are used to test if their concatenations belongs to $\mathcal{U}$. These tests are used to build an observation table.

An *observation table* is a mapping $T : (S \cup (S \times \Sigma)) \times E \to \{true, false\}$ such that for any $s \in S$, $a \in \Sigma$, and $e \in E$, the table entry $T(sae)$ is true if $sae \in \mathcal{U}$. That is the table stores answers of membership queries. This table is classically represented with the values of $S \cup (S \times \Sigma)$ for row entries and the values of E as columns entries.

Definition 3.2. *An observation table is* closed *if the following holds:*

$$\forall s \in S, \forall a \in \Sigma, \exists s' \in S, \forall e \in E : T(sae) = T(s'e)$$

Intuitively, the property says that for every prefix s and letter a there is a prefix s' representing sa.

Definition 3.3. *An observation table is* consistent *if the following holds:*

$$\forall s_1, s_2 \in S, \exists e \in E : T(s_1 e) \neq T(s_2 e)$$

This property says that every two prefixes can be distinguished by some suffix.

Once the observation table is closed and consistent, an automaton is generated that is conjectured to recognize $\mathcal{U}$. This automaton is $\mathcal{C} = \langle Q, \Sigma, \rightarrow, q_0, F \rangle$ where:

- $Q = S$
- $\rightarrow = \{(s, a, s') \in (S \times \Sigma \times S) \mid \forall e \in E, T(sae) = T(s'e)\}$
- $q_0 = \lambda$
- $F = \{s \in S \mid T(s) = true\}$

Since the conjecture is generated from a closed observation table, the existence of a successor for any prefix and any letter is guaranteed. Moreover, since the observation table is consistent there exists a unique successor for each prefix and letters, and therefore the automaton $\mathcal{C}$ is deterministic.

The L^* algorithm works as follows: it starts with the sets S and E containing only the empty word λ. Then it extends the set of prefixes as long as the observation table is not closed. Once the table is closed, it generates a conjecture $\mathcal{C}$ that is passed to the teacher. If the teacher concludes that $\mathcal{C}$ is correct, the algorithm stops and returns $\mathcal{C}$. Otherwise, the counterexample returned by the teacher is analyzed, used to extend E, and the algorithm starts over.

In more details, the L^* algorithm starts by populating its observation table T until it is closed (Lines 3-7 of Figure 3.3). To do so, it looks for a prefix s and a letter a such that no other prefix s' satisfies $T(sae) = T(s'e)$ for all $e \in E$. If no such s and a exists, it follows from Definition 3.2 that the table is closed. Otherwise, the set S is extended with the word sa, updated by membership queries (Lines 8-11), and the loop starts again.

Once the table is closed, a conjecture DFA $\mathcal{C}$ is constructed from T and is passed to the oracle. If the oracle concludes that $\mathcal{C}$ is correct, the algorithm returns Line 16. Otherwise, a counterexample w returned by the oracle is analyzed to extract its longest suffix w' that if added to E would render T not closed. This suffix is added to E, and the algorithm starts over.

Observe that L^* presented here does not require consistency testing of the observation table. This is due to the fact that by construction any prefix added in Line 6 generates a new row whose valuation with respect to E differ from all of the preexisting prefixes.

$L^*(\mathcal{O})$

Input: a Minimal Adequate Teacher $\mathcal{O}$.

```
1  S = E = {λ}
2  while true do
3     while T is not closed do
4        foreach s ∈ S, a ∈ Σ, and e ∈ E do
5           if there does not exists s' ∈ S such that T(sae) = T(s'e) then
6              S ← S ∪ {sa}
7        done
8        foreach s ∈ S, a ∈ Σ, and e ∈ E do
9           Ask O if sae belong to U
10          Update T(sae) with the answer
11       done
12    done
13    Construct C from T
14    Ask O if C is correct
15    if (C is not correct)
16       return C
17    else
18       extract a suffix w' from the counterexample w
19       E ← E ∪ {w'}
20 done
```

Figure 3.3: The L^* algorithm

The suffix added Line 19 obviously cannot render the observation table inconsistent if it was previously consistent.

The algorithm complexity is $\mathbb{O}(n^2|\Sigma| + n \log m)$ where n is the number of states of the output DFA, and m is the size of the longest counterexample.

Learning Assumptions

The assume-guarantee method (see Section 3.3.2) requires the "discovery" of a suitable assumption. This assumption must satisfy the assume-guarantee preconditions: $\langle A \rangle M_1 \langle \varphi \rangle$, and $\langle true \rangle X \langle A \rangle$. To this end, the L^* algorithm has been successfully used [CGP03] to iteratively build assumptions that will discharge the first step of the assume-guarantee rule, and satisfy the second step of the rule.

In this context, the models are viewed as prefix-closed regular languages, and the property φ is a particular regular language. In a language theoretic formulation, the problem is to find an automaton A such that

$$\mathcal{L}(A) \cap \mathcal{L}(M_1) \subseteq \mathcal{L}(\varphi), \quad \text{and} \quad \mathcal{L}(M_2) \subseteq \mathcal{L}(A). \tag{3.1}$$

To this end, the L^* algorithm is used to generate conjectures in order to finally find A satisfying the properties above.

The framework proposed [CGP03] is based on a particular implementation of the oracle used in L^* algorithm. Every conjecture A generated by the L^* algorithm is supposed to be an assumption in the assume-guarantee rule. If it satisfies the two conditions above then the algorithm terminates with success. Otherwise a counterexample is given and analyzed. To some extend, the framework can be viewed as a particular implementation of an L^* with oracle who does not refer to just one unknown language $\mathcal{U}$ but to all assumptions that satisfy the rule. This oracle is implemented as follows:

Membership queries. Membership queries are performed in order to satisfy the first step of the assume-guarantee rule. Hence, the oracle returns true if the word w belongs both to M_1 and φ.

Conjecture testing is performed in the following the sequence of distinct phases:

- The conjecture A is checked to verify if $\langle A \rangle M_1 \langle \varphi \rangle$ holds. If it does not hold, false is returned, and a counterexample is provided to the L^* algorithm.
- The conjecture is now checked against $\langle true \rangle M_2 \langle A \rangle$. If this property holds, the framework returns $M_1 \parallel M_2 \models \varphi$ to the user. Otherwise, π a counterexample to $\langle true \rangle M_2 \langle A \rangle$ is generated and analyzed.
- The counterexample π is analyzed against $\langle A \rangle M_1 \langle \varphi \rangle$. If π is not a word of $\mathcal{L}(M_1) \cap \mathcal{L}(\varphi)$ then, the framework returns $M_1 \parallel M_2 \not\models \varphi$ with π as a counterexample. Otherwise, the conjecture A is rejected, and π is provided to the L^* algorithm.

Remark 3.1. *Here we have skipped over language projection issues (i.e. translating a word of a model to another model using synchronization). This point is not crucial to the understanding of the overall framework.*

Alphabet Refinement.

In the following, we turn our attention to optimization proposed to the learning based assume-guarantee method we have presented above. Note that the models (and the property) in the following are defined over separate alphabets and synchronize on common events.

We now focus on the minimal alphabet needed to define an assumption. Given two models M_1, M_2, and a property φ, the *interface alphabet* for the verification problem $M_1 \parallel M_2 \models \varphi$ is $\Sigma_I = (\Sigma_1 \cup \Sigma_\varphi) \cap \Sigma_2$. This interface alphabet is the maximal subset of the universal alphabet $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_\varphi$ needed to generate an suitable assumption for the assume-guarantee rule. Yet, an assumption may not need the entire interface alphabet to be conclusive.

In the context of learning assumption, improvements to the L^* algorithm have been proposed. *Alphabet refinement* have been introduced to limit the computational cost of the L^* algorithm. Recall that the complexity of the algorithm is $\mathcal{O}(n^2|\Sigma| + n \log m)$ (see Section 3.3.3 for further details). As presented above, the alphabet can be restricted from

the universal alphabet Σ, to the interface alphabet $\Sigma_I = (\Sigma_1 \cup \Sigma_\varphi) \cap \Sigma_2$. This reduces the computational cost of updating the observation table since we use Σ_I instead of Σ. Yet, a method based on an underapproximation of the interface alphabet have been proposed to further reduce the computational cost. *Alphabet Refinement* [GGP07, CS07] techniques manipulate a "smaller" alphabet $\Sigma' \subset \Sigma_I$ as the alphabet of the assumption. Since the interface alphabet is the maximal subset of Σ needed to define the assumption, the objective here is to find a suitable subset of Σ_I.

In more details, the L^* algorithm starts with an empty alphabet and generates an assumption that is tested against the assume-guarantee rule as presented Section 3.3.3. If a counterexample is found, it is analyzed. If the counterexample is due to the coarseness of the used alphabet, the alphabet is refined by adding to it some letters. Various heuristics are proposed in [GGP07, CS07] to select some letters to add to the alphabet. Once the alphabet refined the process starts over.

3.3.4 Abstraction & Assume-Guarantee Reasoning

An abstraction based method have been proposed in the context of assume-guarantee reasoning [BPG08]. The assumption A of the assume guarantee-rule is computed as an abstraction of M_2. Since, the abstraction A of a model M maintains all behaviors of M, the second step of the rule is trivially satisfied (see Section 3.2.1).

In this approach, a partition abstraction of M_2 is computed, and used to verify the first step of the assume-guarantee rule. If the first step of the rule does not hold, a counterexample is generated. This counterexample is then analyzed to determine if it is spurious or not. If the counterexample is not spurious the algorithm stops and returns "false" with the counterexample to the user. Otherwise, the counterexample is used to refine the abstraction A, in order to eliminate this spurious behavior.

Initially, the abstraction is a single state representing all concrete states, and the transition relation is a single "loop" labeled with the alphabet. This abstraction is then used to check the first step of the assume guarantee loop. If the first step is verified, then the framework returns true, otherwise a counterexample is analyzed as follows. The counterexample is simulated on M_2, if it is feasible, then the framework returns false, and provides the counterexample. If spurious, the counterexample is passed together with A and M_2 to a CEGAR procedure that refines A in order to eliminate the counterexample. Once the CEGAR procedure returned a refined abstraction the process starts over.

In [BPG08] alphabet refinement (see Section 3.3.2) methods are also used to reduce the alphabet of the assumption.

3.3.5 Language separation

Language separation [GMF07, CFC+09] is yet another approach to the assumption generation in the assume-guarantee paradigm. Given two disjoint regular languages L_1, and L_2 a separating DFA is a DFA A such that $\mathcal{L}(L_1) \subseteq \mathcal{L}(A)$, and $\mathcal{L}(A) \cap \mathcal{L}(L_2) = \emptyset$. In this context, the assumption A is viewed as a separating language for $\mathcal{L}(M_1)$, and $\mathcal{L}(M_2) \cap \mathcal{L}(\neg\varphi)$.

More precisely, the assumption A is computed so that it satisfies $\mathcal{L}(M_1) \subseteq \mathcal{L}(A)$, and $\mathcal{L}(A) \cap \mathcal{L}(M_2) \cap \mathcal{L}(\neg\varphi) = \emptyset$.

We present here an approach to language separation, proposed by Chen et al. in [CFC+09] that is based on three-valued deterministic finite automata and an algorithm L^{sep}. For notational convenience, we will use the notation $\mathcal{L}(M_2')$ for $\mathcal{L}(M_2) \cap \mathcal{L}(\neg\varphi)$.

Recall that we are here searching for a minimal separating automaton for $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$ when it exists. When such an automaton does not exists we deduce that the assume-guarantee rule cannot be satisfied and conclude that the safety verification failed (and a counterexample is generated).

Three-Valued Deterministic Finite Automata & L^{sep}

A *Three-Valued Deterministic Finite Automata* (3DFA for short), is a tuple $C = \langle Q, \Sigma, \rightarrow, q_0, F, R, D\rangle$, where $\langle Q, \Sigma, \rightarrow, q_0, F\rangle$ is a deterministic finite automata, $R \subseteq Q$ is a set of rejecting states, $D \subseteq Q$ is a set of "don't care" states, and the set $\{F, R, D\}$ is a partition of Q. A word $u \in \Sigma^*$ is accepted by a 3DFA C if the transition sequence $q_0 \xrightarrow{u} q$ satisfies $q \in F$, it is rejected if $q \in R$, and is a don't care string if $q \in D$. The notation C^+ for a 3DFA stands for $C = \langle Q, \Sigma, \rightarrow, q_0, F \cup D\rangle$, and $C^- = \langle Q, \Sigma, \rightarrow, q_0, F, R \cup D\rangle$. A DFA A is *consistent* with a 3DFA C if it satisfies $\mathcal{L}(C^-) \subseteq \mathcal{L}(A) \subseteq \mathcal{L}(C^+)$. A 3DFA C is *sound* with respect to L_1, and L_2 if any DFA consistent with C is a separating DFA for L_1, and L_2. Finally, a 3DFA is *complete* with respect to L_1, and L_2 if any separating DFA for L_1, and L_2 is consistent with C. The problem is here reduced to finding a minimal DFA consistent with a 3DFA which is sound and complete with respect to $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2')$.

The framework proposed in [CFC+09] to compute a separating DFA for two regular languages L_1, and L_2 is composed of the four following steps:

Candidate Generation. A candidate 3DFA C is generated by the L^{sep} algorithm.

Completeness Checking The 3DFA C is tested for completeness. If not complete, a counterexample is returned to the L^{sep} algorithm to get a new candidate.

Minimal Consisting DFA A minimal consisting DFA A is generated from the candidate 3DFA C.

Soundness Checking The DFA A is tested for soundness against L_1, and L_2. If a counterexample is found it is passed to the candidate generator and the process starts over.

We now present in more details these steps. The completeness checking is tested by verifying that $\mathcal{L}(C^-) \subseteq L_1$, and $\neg L_2 \subseteq \mathcal{L}(C^+)$ hold. Soundness checking is tested by verifying that $L_1 \subseteq \mathcal{L}(A)$, and $\mathcal{A} \subsetneq L_2$ hold.

Candidate Generation. An extension of the L^* algorithm is proposed to generate 3DFAs as conjectures: the L^{sep} algorithm. The algorithm extends the classical L^* algorithm with a addition of a third possible value to a word query: the "?" value which stands for "don't care" words. The L^{sep} algorithm goes as L^*: when the observation table is closed and consistent a candidate 3DFA is generated and tested as a conjecture. Counterexamples to the conjecture can come from completeness checking or soundness checking.

Minimal Consistent DFA. Given a complete 3DFA C a consistent DFA A is generated as the third step of the framework. To this end, the 3DFA C is viewed as an incompletely specified machine, and the algorithm proposed in [PU59] is invoked to generate A.

In the context of verification by application of the assume-guarantee rule, the above framework is used to search for a suitable assumption. The problem is therefore reduced to the search of a DFA A such that $\mathcal{L}(M_1) \subseteq \mathcal{L}(A) \subseteq \neg(\mathcal{L}(M_2) \cap \mathcal{L}(\neg\varphi))$. Note that the algorithm is modified to return "fail" when the property is violated. The modifications are:

- While performing a membership query, if a word w belongs both to L_1, and L_2 then the algorithm returns "fail" together with the word w.
- When a counterexample w is returned to the L^{sep} algorithm, the counterexamples is passed to the oracle for a membership query as described above.

Note that, Gupta et al. in [GMF07] proposed another language separation approach based on incomplete deterministic finite automaton.

Chapter 4

CEGAR with Pruning

Introduction

Abstract counterexample analysis and refinement often requires the computation of precise and costly reachability information on the concrete model. In this chapter, we show how to enrich and exploit this reachability information in order to improve the classical CEGAR paradigm for safety verification. To this end, we introduce the notion of *certified approximation*: an extension of standard existential abstraction where some abstract states are additionally identified as containing only reachable (or co-reachable) concrete states. The main contribution here is a reduction method, based on certified approximations, to identify useless abstract states and prune them from the abstraction. This pruning reduces the set of abstract states, which reduces the computational resources (time and memory) required to build and explore the abstraction. Moreover, it also helps to avoid useless refinements (which in turn accelerates the CEGAR loop), and it focuses the algorithm on counterexamples that are shorter and more likely to be feasible. Certified approximations are not conservative in the classical sense, since their abstract state space is an under-approximation of the concrete state space. However, we show that it sound to use them in a CEGAR algorithm for safety verification. In Chapter 6 we describe experiments performed with our implementation of this approach on top of the BDD-based Mec 5 model-checker. The results confirmed the above-mentioned expected improvements.

Motivating Example. Consider the program with control flow graph given in Figure 4.1(a). The variables `x` and `y` range over $\mathbb{Z}$. Its set of (control) locations is $L = \{A, \ldots, G\}$, its initial location is A and its error location is G. We assume a standard operational semantics for the program, given as a transition system S whose state set Q is the set of triples (l, x, y) where $l \in L$ is a location and $(x, y) \in \mathbb{Z}^2$ is a valuation of the program's variables, and whose transition relation is induced by the program statements. The set of initial states of S is $\{A\} \times \mathbb{Z}^2$. Here, we want to verify that no state in $\{G\} \times \mathbb{Z}^2$

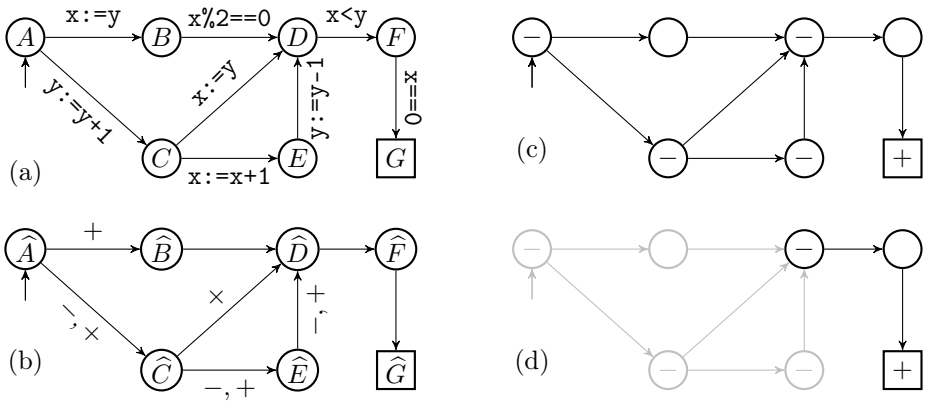


Figure 4.1: Motivating example: (a) control flow graph, (b) initial partition abstraction and must transitions, (c) resulting certified abstraction, (d) certified approximation after pruning.

is reachable.

Let us apply the classical CEGAR approach [CGJ+03] to check whether the error location G is reachable or not. We choose as initial abstraction the partition $\widehat{Q} = \{\widehat{A}, \widehat{B}, \ldots, \widehat{G}\}$ of Q induced by the control locations ($\widehat{l} = \{l\} \times \mathbb{Z}^2$ for each $l \in L$). The CEGAR algorithm explores this abstraction (in a breadth-first manner) and returns, for instance, the abstract counterexample $\widehat{A}\,\widehat{C}\,\widehat{D}\,\widehat{F}\,\widehat{G}$. This counterexample is obviously spurious, and the abstraction is refined by splitting the abstract state $\widehat{D}$ into $\widehat{D}^{=} = \{(D, x, y) \mid x = y\}$ and $\widehat{D}^{\neq} = \widehat{D} \setminus \widehat{D}^{=}$. Iterating the CEGAR loop, the next counterexample is $\widehat{A}\,\widehat{B}\,\widehat{D}^{\neq}\,\widehat{F}\,\widehat{G}$. Likewise, it is spurious, and the abstraction is refined by splitting the abstract state $\widehat{B}$ into $\widehat{B}^{=}$ and $\widehat{B}^{\neq}$. Finally, the third counterexample is $\widehat{A}\,\widehat{C}\,\widehat{E}\,\widehat{D}^{\neq}\,\widehat{F}\,\widehat{G}$, which is feasible, and the algorithm returns that G is reachable.

The approach that we propose in this chapter performs a pruning step at each CEGAR iteration to reduce the abstract state space. To this end, we identify a *certified pair* $(\widehat{Q}_-, \widehat{Q}_+)$ of sets of abstract states that contain only reachable and co-reachable concrete states, respectively. In our example, we can obviously put $\widehat{A}$ into $\widehat{Q}_-$ and $\widehat{G}$ into $\widehat{Q}_+$. Must transitions ($\xrightarrow{+}$) and their dual ($\xrightarrow{-}$) can be used to enlarge certified pairs. The $\xrightarrow{+}$ transition proposed by Larsen in [Lar89], and used in the context of abstraction by Shoham et al in [SG04], identifies a reachability property between two abstract states. When we have a $\xrightarrow{+}$ transition between two abstract states: $\widehat{q} \xrightarrow{+} \widehat{r}$, it follows that each state abstracted by $\widehat{q}$ is a predecessor of some state abstracted by $\widehat{r}$. Put differently, we say that the states of $\widehat{q}$, are co-reachable from the states of $\widehat{r}$. The dual must transition $\xrightarrow{-}$ proposed by Ball et al. in [BKY05] uses the dual property: each state of $\widehat{r}$ is reachable from some state of $\widehat{q}$. Figure 4.1(b) depicts these must transitions for the initial partition abstraction. We deduce from them that $\widehat{C}, \widehat{E}$ and $\widehat{D}$ can be added to $\widehat{Q}_-$. The resulting *certified abstraction* (i.e., the abstraction equipped with this certified pair) is presented in Figure 4.1(c). It is now clear that this certified abstraction can be reduced to the abstract states $\widehat{D}, \widehat{F}, \widehat{G}$ without any loss of precision regarding the reachability question.

Thus, we remove all other abstract states, and obtain the *certified approximation* depicted in Figure 4.1(d). After this pruning, our algorithm performs a classical CEGAR step (and then jumps back to pruning). The exploration of the certified approximation returns the counterexample $\widehat{D}\,\widehat{F}\,\widehat{G}$, which is feasible, and our algorithm returns that G is reachable. This example demonstrates the expected benefits of our approach: the abstract state space is reduced, counterexamples are shorter and have better chances of proving unsafety. Experimental results confirm these expectations.

This notion of abstraction pruning has been studied from different angles. Path slicing [JM05] is a well-known, and widely used static analysis method that prunes away irrelevant parts of a program path when checking its feasibility. More recently, a related method has been proposed for abstractions [BDFW08] in order to reduce and simplify their analysis in a CEGAR fashion. In more detail, a data-flow analysis is performed to identify a set of live variables which are used to optimize the feasibility checking [JM05], and the computation of the abstract transition relation [BDFW08]. These slicing methods are syntax-based, whereas our pruning technique is semantics-based, and, thus can be applied to a wider collection of model representations.

Must transitions ($\xrightarrow{+}$) and their dual ($\xrightarrow{-}$) have been used by Ball et al. in the context of LTL model-checking with cartesian predicate abstraction [BKY05] (see Section 3.2.1). In particular, they obtain a sufficient condition for unsafety based on these must transitions. We also use these must transitions, but our objective is different: we exploit them to enlarge certified pairs, which leads to better state space reduction. Moreover, our algorithm tests a sufficient condition for unsafety based on certified pairs as well as must transitions. This sufficient condition subsumes the one proposed in [BKY05].

The chapter is organized as follows. In Section 4.1, we define cover abstractions of transition systems. Section 4.2 presents certified approximations, and our reduction technique. Section 4.3 discusses inference methods for certified pairs. Our pruning-based CEGAR algorithm is the focus of Section 4.4. Section 4.5 presents results on the optimality of our pruning-based CEGAR algorithm, and some experimental results are presented in Section 4.6. Conclusions and perspectives are given in Section 4.7.

4.1 Transition Systems and Cover Abstractions

This section presents basic definitions and recalls the main concepts underlying counter-example guided abstraction refinement [CGJ+03]. Instead of working with partitions of the state space, we prefer a more general setting using covers.

4.1.1 Cover Abstractions

A *cover* of a set A is a subset $C \subseteq \mathcal{P}^+(A)$ of nonempty subsets of A, such that $A = \bigcup C$. Put differently a cover of a set is a collection of its subsets whose union forms the set. Here, our intent is to use covers of the state space in order to define conservative abstractions that are used to solve the safety verification problem.

The safety verification problem that we address in this chapter amounts to checking, for a given transition system $S = \langle Q, \rightarrow, I, F \rangle$, the emptiness of the set $Run(S)$ of all runs of S, where F is the set of "bad" states (see Definition 1.1 and Definition 1.2). We will present two (semi)-algorithms that take as input a transition system S and decide whether $Run(S)$ is empty or not. These algorithms return either "$Run(S) = \emptyset$" or "$Run(S) \neq \emptyset$" when they terminate, and they will be called *correct* if the returned answer (if any) is always correct.

Definition 4.1. *Let $S = \langle Q, \rightarrow, I, F \rangle$ be a transition system. Given $\widehat{Q} \subseteq \mathcal{P}^+(Q)$, the* approximation *of S induced by $\widehat{Q}$ is the transition system $S[\widehat{Q}] = \langle \widehat{Q}, \rightarrow, \widehat{I}, \widehat{F} \rangle$ defined by:*

$$\begin{cases} \widehat{q} \rightarrow \widehat{r} & \Leftrightarrow & \rightarrow \cap (\widehat{q} \times \widehat{r}) \neq \emptyset \\ \widehat{q} \in \widehat{I} & \Leftrightarrow & \widehat{q} \cap I \neq \emptyset \\ \widehat{q} \in \widehat{F} & \Leftrightarrow & \widehat{q} \cap F \neq \emptyset \end{cases}$$

If $\widehat{Q}$ is a cover of Q, we call $S[\widehat{Q}]$ a cover abstraction *of S.*

In the remainder of this chapter, we fix a transition system $S = \langle Q, \rightarrow, I, F \rangle$. By a slight abuse of notation, we will simply write $Run(\widehat{Q})$ for $Run(S[\widehat{Q}])$. To prevent confusion, states, transitions, and paths of S or $S[\widehat{Q}]$ will be called *concrete* or *abstract*, respectively.

A cover abstraction $S[\widehat{Q}]$ is an *abstraction* of S in the classical sense, i.e. there exists a simulation relation [Mil71] from S to $S[\widehat{Q}]$ that maps the initial states and final states of S to the initial states and final states of $S[\widehat{Q}]$, respectively. Conversely, every simulation relation-based abstraction can be viewed as a cover abstraction (but not necessarily as a partition abstraction). Thus our definition captures abstract interpretation-based abstractions [DGG97], such as cartesian predicate abstractions [GS97].

The *concretization* of an abstract path $\widehat{\pi} = \widehat{q}_0, \ldots, \widehat{q}_n$ in an approximation $S[\widehat{Q}]$ is the set of all finite sequences $q_0, \ldots, q_n$ of concrete states such that $q_i \in \widehat{q}_i$ for all $0 \leq i \leq n$. Notice that the concretization of an abstract path may contain sequences of concrete states that are not concrete paths. An abstract run $\widehat{\pi}$ is called *feasible* if its concretization contains a run of S, and is called *spurious* otherwise. Observe that if $S[\widehat{Q}]$ is a cover abstraction, then every run of S is in the concretization of some abstract run.

4.1.2 Cover Abstractions and CEGAR

Equipped with our abstraction method, we now turn our attention to the next step of the CEGAR [CGJ+03] loop: refinement of abstractions. Various refinements techniques have been proposed (see Chapter 3.2.3) to eliminate spurious counterexamples. This chapter investigates improvements of the CEGAR loop that are orthogonal to classical refinement techniques. Hence, we adopt a generic and abstract view of refinement: we see refinement as a black-box operation that splits an abstract state into several smaller ones. Formally, given an approximation $S[\widehat{Q}]$, a *split* is a pair $(\widehat{x}, \widehat{X})$ in $(\widehat{Q} \times \mathcal{P}(\mathcal{P}^+(Q)))$ such that $\widehat{x} \notin \widehat{X}$ and $\widehat{x} = \bigcup \widehat{X}$.

We are now equipped with the main ingredients to present Cegar, the classical CEGAR algorithm. In Section 3.2.4 we have presented the CEGAR method as a parameterized algorithm where choices had to be made to select an abstraction method, a model checker (or at least a model checking technique), an abstract counter-example verification method, and refinement method. Here, we present a CEGAR algorithm that uses cover abstraction, and requires a decision procedure that determines the emptiness of the set of runs of the abstraction. Note that we still leave unspecified the refinement method and the abstract counter-example verification method, as they are not the subject of this chapter.

The Cegar algorithm starts with an initial cover abstraction, e.g., the one induced by the cover $\{Q\}$ of the set Q of concrete states. At each iteration of the while loop, an abstract run $\widehat{\pi}$ is picked in the abstraction. If $\widehat{\pi}$ is feasible, i.e., its concretization contains a concrete run, then Cegar returns "$Run(S) \neq \emptyset$". Otherwise, $\widehat{\pi}$ is spurious, and a refinement is performed (lines 6–7) on the abstraction (a priory to eliminate $\widehat{\pi}$). Note that the existence of a split pair is guaranteed, since $\widehat{\pi}$ is spurious. This process is iterated until no abstract run remains in $S[\widehat{Q}]$, in which case "$Run(S) = \emptyset$" is returned. Even though our theoretical approach does not require it, in practice, implementations of Cegar choose a split (at line 6) that ensures elimination of the spurious abstract run $\widehat{\pi}$.

It is readily seen that, at each iteration of the while loop, $S[\widehat{Q}]$ is a cover abstraction. This entails the correctness of Cegar. However, termination of Cegar for finite transition systems is less obvious. Indeed, since we work with covers instead of partitions, an abstract state that is split and removed may appear again at a later iteration (as part of $\widehat{X}$). Still, if Q is finite, each abstract state may only reappear finitely many times. We obtain the following proposition.

Proposition 4.1. *Given a cover $\widehat{Q} \subseteq \mathcal{P}^+(Q)$, $\mathsf{Cegar}(S, \widehat{Q})$ is correct, and it terminates if Q is finite.*

Proof. Let us first prove correctness. Is is routinely checked, by induction, that $\widehat{Q}$ is a cover of Q at each iteration of the while loop. If $\mathsf{Cegar}(S, \widehat{Q})$ returns "$Run(S) \neq \emptyset$" from Line 4, then the abstract run $\widehat{\pi}$ is feasible, hence, there exists a run in S. Suppose, on the contrary, that $\mathsf{Cegar}(S, \widehat{Q})$ returns "$Run(S) = \emptyset$" from Line 9. The while loop condition

$\mathsf{Cegar}\,(S, \widehat{Q})$
Input: a transition system S, a cover $\widehat{Q}$ of Q.

```
1 while Run(Q̂) ≠ ∅ do
2    Pick an abstract run π̂ in Run(Q̂)
3    if π̂ is feasible then
4       return "Run(S) ≠ ∅"
5    else  // π̂ is spurious
6       Pick a split (x̂, X̂)
7       Q̂ ← (Q̂ \ x̂) ∪ X̂
8 done
9 return "Run(S) = ∅"
```

is not satisfied, hence, $Run(\widehat{Q}) = \emptyset$. This entails that $Run(S) = \emptyset$ since every run of S is in the concretization of some abstract run in $Run(\widehat{Q})$.

To prove termination, assume that Q is finite. We introduce the ranking function $f : \widehat{Q} \to \mathbb{N}^{|Q|}$ defined by: $f(\widehat{Q}) = (c_{|Q|}, \ldots, c_1)$ where $c_i = |\{\widehat{q} \in \widehat{Q} \mid |\widehat{q}| = i\}|$. Let $\leq$ denote the usual lexicographic order over $\mathbb{N}^{|Q|}$. We prove that $f(\widehat{Q}') < f(\widehat{Q})$ when $\widehat{Q}' = (\widehat{Q} \setminus \widehat{x}) \cup \widehat{X}$ is the refinement of $\widehat{Q}$ induced by a split $(\widehat{x}, \widehat{X})$. By definition of splits, since $\widehat{x} \notin \widehat{X}$ and $\widehat{x} = \bigcup \widehat{X}$, it holds that $|\widehat{y}| < |\widehat{x}|$ for every $\widehat{y} \in \widehat{X}$. Let us write $f(\widehat{Q}) = (c_{|Q|}, \ldots, c_i, \ldots, c_1)$, where $i = |\widehat{x}|$, and $f(\widehat{Q}') = (c'_{|Q|}, \ldots, c'_i, \ldots, c'_1)$. We get that $c'_i = c_i - 1$ and, for every j from $|Q|$ to $i+1$, $c'_j = c_j$. Thus $f(\widehat{Q}') < f(\widehat{Q})$. We have shown that $f(\widehat{Q})$ strictly decreases at each iteration of the while loop. Since $\leq$ is well-founded on $\mathbb{N}^{|Q|}$, we derive that $\mathsf{Cegar}(S, \widehat{Q})$ terminates. □

4.2 Pruning of Cover Abstractions

In this chapter we modify the classical Cegar algorithm by storing some additional information about reachable and co-reachable states (cf. the notion of certified pair). We will show that this information, that is computed by classical CEGAR tools anyway, can be used to speed-up the Cegar loop. In particular, termination can be detected sooner, and some useless abstract states can be pruned from the abstraction.

4.2.1 Certified Pairs & Certified Approximations

Definition 4.2. *Given a subset $\widehat{Q}$ of $\mathcal{P}^+(Q)$, a* certified pair *(for $\widehat{Q}$) is a pair $(\widehat{Q}_-, \widehat{Q}_+)$ of subsets of $\widehat{Q}$ satisfying:*

- *if $\widehat{q} \in \widehat{Q}_-$ then $\widehat{q} \subseteq post^*_S(I)$,*
- *if $\widehat{q} \in \widehat{Q}_+$ then $\widehat{q} \subseteq pre^*_S(F)$.*

The triple $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ is called a certified approximation.

In a CEGAR context, we suppose that the $post^*$ and pre^* are prohibitively expensive, this is why we only have implications and not equivalences in the definition of certified pairs. In particular, the pair of empty sets $(\emptyset, \emptyset)$ is a certified pair. We will see in Section 4.3 how to populate certified pairs. Two methods will be proposed. The first one is based on approximation analysis, namely closure under must transitions. The second one relies on spurious abstract run analysis, and will be discussed in the context of two well-known refinement heuristics.

The first advantage of certified approximations is that they allow to conclude existence of a run in an easy way.

Proposition 4.2. *Let $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ be a certified approximation. The set $Run(S)$ is non-empty if the following condition holds:*

$$\widehat{I} \cap \widehat{Q}_+ \neq \emptyset \text{ or } \widehat{F} \cap \widehat{Q}_- \neq \emptyset \text{ or } \widehat{Q}_- \cap \widehat{Q}_+ \neq \emptyset \text{ or } (\widehat{Q}_- \times \widehat{Q}_+) \cap \to \neq \emptyset \quad (\mathcal{W})$$

Proof. Recall that $\widehat{q}$ is non-empty for every $\widehat{q} \in \widehat{Q}$. If $\widehat{I} \cap \widehat{Q}_+ \neq \emptyset$ then there exists $\widehat{q} \in \widehat{I}$ such that $\widehat{q} \subseteq pre^*_S(F)$. This entails that I intersects $pre^*_S(F)$, hence, $Run(S) \neq \emptyset$. The proof that $\widehat{F} \cap \widehat{Q}_- \neq \emptyset$ implies $Run(S) \neq \emptyset$ is similar. If $\widehat{Q}_- \cap \widehat{Q}_+ \neq \emptyset$ then $post^*_S(I)$ and $pre^*_S(F)$ intersect, hence, $Run(S) \neq \emptyset$. For the last case, suppose that $(\widehat{Q}_- \times \widehat{Q}_+)$ has a non-empty intersection with the abstract transition relation $\rightarrow$. There exists $\widehat{q}_- \in \widehat{Q}_-$ and $\widehat{q}_+ \in \widehat{Q}_+$ such that $\widehat{q}_- \rightarrow \widehat{q}_+$. By Definition 4.1, we obtain that $q_- \rightarrow q_+$ for some concrete states $q_- \in \widehat{q}_-$ and $q_+ \in \widehat{q}_+$. This entails that $Run(S) \neq \emptyset$ since $q_- \in post^*_S(I)$ and $q_+ \in pre^*_S(F)$. □

4.2.2 Kernel Paths

The second and main advantage of certified approximations is that they allow to introduce a stricter notion of abstract run, and, in consequence, eliminate states that are not on these runs.

Definition 4.3. *Given a certified approximation $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$, a* kernel path *is an abstract path $\widehat{\pi} = \widehat{q}_0, \ldots, \widehat{q}_n$ satisfying the following property:*

$$\widehat{q}_0 \in (\widehat{I} \cup \widehat{Q}_-) \wedge \widehat{q}_n \in (\widehat{F} \cup \widehat{Q}_+) \wedge \bigwedge_{i=1}^{n-1} \widehat{q}_i \notin (\widehat{Q}_- \cup \widehat{Q}_+)$$

We write $KerPath(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ for the set of all kernel paths.

Observe that $Run(\widehat{Q}) = KerPath(\widehat{Q}, \emptyset, \emptyset)$. Compared to runs, kernel paths can start from states in $\widehat{Q}_-$ and end in states of $\widehat{Q}_+$. The last condition not only enforces the absence of redundant parts in a kernel path, but also permits state pruning as described later.

We now explain why it is enough to look at kernel paths instead of abstract runs. But first, feasibility of abstract runs must be generalized to kernel paths. A kernel path $\widehat{q}_0, \ldots, \widehat{q}_n$ is *feasible* if there exists a path $q_0, \ldots, q_n$ in S satisfying:

$$(\widehat{q}_0 \notin \widehat{Q}_- \Rightarrow q_0 \in I) \wedge (\widehat{q}_n \notin \widehat{Q}_+ \Rightarrow q_n \in F) \wedge \bigwedge_{i=0}^{n} q_i \in \widehat{q}_i$$

Recall that the definition of kernel paths requires that $\widehat{q}_0$ belongs to $\widehat{I}$ or $\widehat{Q}_-$. In the above condition for feasibility, we ask, in addition, that $q_0 \in I$ when $\widehat{q}_0 \notin \widehat{Q}_-$. This comes from Definition 4.1: $\widehat{q}_0 \in \widehat{I}$ only guarantees that $\widehat{q}_0 \cap I \neq \emptyset$, whereas $\widehat{q}_0 \in \widehat{Q}_-$ entails that $\widehat{q}_0 \subseteq post^*_S(I)$. Therefore, we ask that $q_0 \in I$ in order to be able to extract a concrete run from a feasible kernel path. For the same reason, we ask that $q_n \in F$ when $\widehat{q}_n$ does not belong to $\widehat{Q}_+$. The extraction of concrete runs from feasible kernel paths is formalized in the following lemma.

Lemma 4.1. *Let $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ be a certified approximation. If there exists a feasible kernel path in $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ then $Run(S) \neq \emptyset$.*

Proof. Assume that $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ contains a feasible kernel path $\widehat{q}_0, \ldots, \widehat{q}_n$. By definition, there exists a path $q_0, \ldots, q_n$ in S such that $\widehat{q}_0 \notin \widehat{Q}_- \Rightarrow q_0 \in I$, $\widehat{q}_n \notin \widehat{Q}_+ \Rightarrow q_n \in F$, and $q_i \in \widehat{q}_i$ for all $i \in \{0, \ldots, n\}$. The first condition entails that $q_0 \in post^*_S(I)$. Likewise, the second condition entails that $q_n \in pre^*_S(F)$. Since $q_0, \ldots, q_n$ is a path in S, we conclude that $Run(S) \neq \emptyset$. □

The previous lemma provides a sufficient condition for non-emptiness of $Run(S)$, based on feasibility of kernel paths. The converse does not hold, in general, for certified approximations. But in the particular case of certified cover abstractions, we obtain a necessary and sufficient condition.

Remark 4.1. *Let $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ be a certified cover abstraction, i.e., a certified approximation where $\widehat{Q}$ is a cover. It is routinely checked that every kernel path is a factor of some abstract run, and, conversely, every abstract run contains a kernel path among its factors. We obtain that $Run(S) = \emptyset$ if and only if $KerPath(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+) = \emptyset$.*

The previous remark implies that it is enough to consider states that appear on kernel paths.

4.2.3 Kernel States

Definition 4.4. *Given a certified approximation $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$, a* kernel state *is an abstract state occurring on some kernel path. The set of all kernel states is denoted by $Ker(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$.*

It is not difficult to compute kernel states. We can compute the set of states A reachable in $S[\widehat{Q}]$ from $\widehat{I} \cup \widehat{Q}_-$ without going through a state of $\widehat{Q}_+$; and the set of states B co-reachable from $\widehat{F} \cup \widehat{Q}_+$ without passing through $\widehat{Q}_-$. This can be done in linear time using a simple graph exploration of the, finite, transition system $S[\widehat{Q}]$. We then derive $Ker(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+) = A \cap B$.

It follows from Remark 4.1 that, in a certified cover abstraction, it is sufficient to analyze kernel paths. Therefore, we can safely restrict the approximation to its kernel states.

Definition 4.5. *The* reduction *of a certified approximation $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$, denoted by $Red(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$, is the certified approximation $(\widehat{Q}^r, \widehat{Q}^r_-, \widehat{Q}^r_+)$ where $\widehat{Q}^r = Ker(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$, $\widehat{Q}^r_- = \widehat{Q}_- \cap \widehat{Q}^r$ and $\widehat{Q}^r_+ = \widehat{Q}_+ \cap \widehat{Q}^r$.*

As an illustration of the reduction operation, consider our example given in Figure 4.1. A certified cover abstraction $(\widehat{Q} = \{\widehat{A}, \widehat{B}, \widehat{C}, \widehat{D}, \widehat{E}, \widehat{F}, \widehat{G}\}, \widehat{Q}_- = \{\widehat{A}, \widehat{C}, \widehat{D}, \widehat{E}\}, \widehat{Q}_+ = \{\widehat{G}\})$ of the program is given in Figure 4.1(c). Its unique kernel path is $\widehat{D}\,\widehat{F}\,\widehat{G}$, thus its kernel states are $\widehat{D}, \widehat{F}$ and $\widehat{G}$. The reduction of this cover abstraction is the certified approximation shown in Figure 4.1(d).

Our intention is to apply reduction at each iteration of the Cegar loop. However, after an application of reduction, the set of abstract states may not be a cover anymore, and,

therefore, Remark 4.1 cannot be applied. We will show that the certified approximations computed by our upcoming PCegar algorithm are *complete*, in the sense that they still satisfy the property of Remark 4.1 even though they are not cover abstractions.

4.3 Inference of Certified Pairs

In this section we present different methods to extend certified pairs by means of approximation analysis, and refinement.

4.3.1 Abstraction & Must Transitions

One way to extend certified pairs is to use *must* transitions in addition to the preexisting *may* transitions. We write

$$\widehat{q} \xrightarrow{+} \widehat{r} \Leftrightarrow \widehat{q} \subseteq pre_S(\widehat{r}).$$

for a must transition between two abstract states. We will also need dual transitions:

$$\widehat{q} \xrightarrow{-} \widehat{r} \Leftrightarrow \widehat{r} \subseteq post_S(\widehat{q})$$

Observe that, since we impose $\emptyset \notin \widehat{Q}$, each must transition is a may transition: $\xrightarrow{-} \subseteq \rightarrow$ and $\xrightarrow{+} \subseteq \rightarrow$. These notions allow us to enlarge the sets $\widehat{Q}_-$ and $\widehat{Q}_+$ by taking their closure under appropriate transitions.

Definition 4.6. *Given a certified approximation* $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$*, the* closure *of* $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$*, written* $Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$*, is the tuple* $(\widehat{Q}^c, \widehat{Q}^c_-, \widehat{Q}^c_+)$ *where:*

$$\begin{cases} \widehat{Q}^c &= \widehat{Q} \\ \widehat{Q}^c_- &= (\xrightarrow{-})^*[\widehat{Q}_- \cup (\widehat{I} \cap \mathcal{P}(I))] \\ \widehat{Q}^c_+ &= \left((\xrightarrow{+})^{-1}\right)^* [\widehat{Q}_+ \cup (\widehat{F} \cap \mathcal{P}(F))] \end{cases}$$

By definition, it is clear that the closure of a certified approximation is a certified approximation.

As an illustration of the closure operation, consider the cover abstraction depicted in Figure 4.1(b). Pick for instance the abstract transition $\widehat{A} \rightarrow \widehat{C}$. Clearly, for all $(x, y) \in \mathbb{Z}^2$, it holds that (x, y) is a successor of $(x, y-1)$ under the assignment $\mathtt{y} := \mathtt{y}+\mathtt{1}$. This entails that $\widehat{C} \subseteq post_S(\widehat{A})$, hence, $\widehat{A} \xrightarrow{-} \widehat{C}$. Similarly, we get that there are 8 must transitions: $\xrightarrow{-} = \{(\widehat{A}, \widehat{C}), (\widehat{C}, \widehat{E}), (\widehat{E}, \widehat{D})\}$ and $\xrightarrow{+} = \{(\widehat{A}, \widehat{B}), (\widehat{A}, \widehat{C}), (\widehat{C}, \widehat{D}), (\widehat{C}, \widehat{E}), (\widehat{E}, \widehat{D})\}$. The application of closure leads to the certified cover abstraction depicted in Figure 4.1(c).

Remark 4.2. *Testing the* $\mathcal{W}$ *condition of Proposition 4.2 after an application of closure captures the sufficient condition for unsafety presented in [BKY05]. In our setting, this condition can be expressed as follows:* $Run(S) \neq \emptyset$ *if there exists an abstract state* $\widehat{q}$

that is reachable from $\widehat{u} \subseteq I$ using $\xrightarrow{-}$ transitions, and co-reachable from $\widehat{v} \subseteq F$ using $\xrightarrow{+}$ transitions. Clearly, the abstract state $\widehat{q}$ belongs to $\widehat{Q}^c_- \cap \widehat{Q}^c_+$, hence, the closure of the certified approximation satisfies the $\mathcal{W}$ condition.

4.3.2 Refinement of a Certified Approximation

We have seen how to populate certified pairs by a graph exploration based on must transitions. However, this may not be sufficient in practice since there is no guarantee that (useful) must transitions exist in the approximation. Therefore, we now propose methods to enlarge certified pairs during the refinement process. Indeed, in a CEGAR approach, the analysis of an abstract run often involves the computation of the iterated concrete *post* or *pre* operation along it. When the abstract run is spurious, the "failure state" (i.e., the abstract state that will be split) holds information about $post^*_S(I)$ or $pre^*_S(F)$. This information is usually discarded, which is a pity since it is both costly and precise. We demonstrate, in the context of two well-known refinement schemes, how to use this information to extend certified pairs. To this end, the notion of split is extended with two sets of abstract states that are suitable to enlarge certified pairs.

Definition 4.7. *Given a certified approximation $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$, a certified split is a tuple $(\widehat{x}, \widehat{X}, \widehat{X}_-, \widehat{X}_+)$ where $(\widehat{x}, \widehat{X})$ is a split, and $(\widehat{X}_-, \widehat{X}_+)$ is a certified pair for $\widehat{Q}$.*

Let us point out that in the definition of a certified split, we do not impose $\widehat{X}_- \subseteq \widehat{X}$ nor $\widehat{X}_+ \subseteq \widehat{X}$. This allows us to take into account reachability (or co-reachability) information that can be extracted during the refinement process. For instance, during the analysis of an abstract run, the iterated concrete *post* (or *pre*) operation can identify some abstract states that only contain reachable (or co-reachable) concrete states. These abstract states can then be added to the set $\widehat{X}_-$ (or $\widehat{X}_+$).

Observe that for every split $(\widehat{x}, \widehat{X})$ the tuple $(\widehat{x}, \widehat{X}, \emptyset, \emptyset)$ is a certified split. We first show how to augment a certified split from the preexisting information held by the certified approximation. Given a certified approximation $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$, we can extend a certified split $(\widehat{x}, \widehat{X}, \widehat{X}_-, \widehat{X}_+)$ by adding the set $\widehat{Y}_- = \{\widehat{y} \in \widehat{X} \mid \widehat{y} \subseteq \bigcup \widehat{Q}_-\}$ to $\widehat{X}_-$, and the set $\widehat{Y}_+ = \{\widehat{y} \in \widehat{X} \mid \widehat{y} \subseteq \bigcup \widehat{Q}_+\}$ to $\widehat{X}_+$. In general the computation of these sets could be expensive. Let us remark though that we can obtain a weaker certified split at no cost by simply adding $\widehat{X}$ to $\widehat{X}_-$ (resp. to $\widehat{X}_+$) if $\widehat{x}$ belongs to $\widehat{Q}_-$ (resp. to $\widehat{Q}_+$). This weaker extension of certified splits is enough, and in fact necessary (as we must not lose certified states by refinement), to ensure correctness of our upcoming PCegar algorithm (it is performed at lines 15–16). We also observe that, when $\widehat{Q}$ is a partition, this weaker extension is equivalent to the previously proposed full extension with the sets $\widehat{Y}_-$ and $\widehat{Y}_+$.

The above extensions of certified splits only use the information that is already available in the certified approximation. We now discuss the enlargement of certified splits in the context of two well-known refinement schemes. We do not present all details of each scheme, and we limit ourselves to our simpler setting of safety verification. For more details, the reader is referred to the respective papers.

First, let us start with the original CEGAR refinement scheme proposed by Clarke et al. [CGJ+03]. In their setting, the abstraction is induced by an equivalence relation over the concrete states (i.e., they consider cover abstractions where the cover is a partition). The abstract counterexample $\widehat{\pi}$ (picked at line 3 of our Cegar algorithm) is analyzed by an iterated concrete *post* computation along the abstract path (algorithm SplitPath in [CGJ+03]). If $\widehat{\pi}$ is spurious, a "failure state" $\widehat{x}$ occurring in $\widehat{\pi}$ is identified together with two disjoint subsets $B, D \subseteq \widehat{x}$ where B is the set of "bad" states, and D is the set of "dead-end" states. The set of "dead-end" states is the last non-empty set of concrete states computed by the SplitPath algorithm, and the set of "bad" states is the set of concrete predecessors in $\widehat{x}$ of the next abstract state in $\widehat{\pi}$. Then the abstraction is refined by a split pair that separates B from D (algorithm PolyRefine in [CGJ+03]). In fact, the implementation reported in [CGJ+03] uses a heuristic that simply refines $\widehat{x}$ with the split $(\widehat{x}, \{D, \widehat{x} \setminus D\})$. Notice that $D \subseteq post^*_S(I)$. Therefore, in our setting, we obtain the certified split $(\widehat{x}, \{D, \widehat{x} \setminus D\}, \{D\}, \emptyset)$. We use this scheme in our implementation to obtain certified splits.

Now consider the refinement scheme proposed by Shoham et al. for CTL model-checking [SG04]. In their setting, the abstraction is induced by a total concretization function mapping each abstract state to a set of concrete states (i.e., they consider cover abstractions). Moreover, their abstraction carries both may transition as well as must transitions (the $\xrightarrow{+}$ transitions that we use for the closure operation). Let us summarize the verification of the CTL formula $EFerror$ following the approach of [SG04]. To fit our setting let F denote the set of concrete states that satisfy *error*, and define $\widehat{F}_+$ as the set of abstract states that are contained in F. First, they compute the set $\widehat{F}^*_+$ of co-reachable abstract states from $\widehat{F}_+$ via must transitions. If $\widehat{F}^*_+$ contains a concrete initial state then $EFerror$ holds. In our setting, the $\mathcal{W}$ condition is satisfied. Otherwise, if $\widehat{F}^*_+$ cannot be extended using may transitions, then $EFerror$ does not hold. Likewise, in our setting, the set of kernel states becomes empty after closure. If no conclusive answer was obtained, a "may-predecessor" of some abstract state in $\widehat{F}^*_+$ is split so as to introduce a new must transition to an abstract state in $\widehat{F}^*_+$. In other worlds, an abstract state $\widehat{x}$ is split in order to introduce a set $B \subseteq \widehat{x}$ that satisfies $B \xrightarrow{+} \widehat{q}$ with $\widehat{q} \in \widehat{F}^*_+$, and the split is $(\widehat{x}, \{B, \widehat{x} \setminus B\})$. Since $\widehat{q}$ belongs to $\widehat{F}^*_+$ we have $\widehat{q} \subseteq pre^*_S(F)$. Hence, by definition of $\xrightarrow{+}$, we get $B \subseteq pre^*_S(F)$. Therefore, in our setting, we obtain the certified split $(\widehat{x}, \{B, \widehat{x} \setminus B\}, \emptyset, \{B\})$.

4.4 CEGAR with Abstraction Pruning

We now present our PCegar algorithm, that extends the classical CEGAR paradigm with abstract state pruning. This pruning not only reduces the computational resources to maintain and explore the abstraction, but also leads to fewer refinements and focuses the algorithm on abstract counterexamples that are shorter and more likely to be feasible.

PCegar $(S, \widehat{Q})$
Input: a transition system S, a cover $\widehat{Q}$ of Q.

```
1  (Q̂₋, Q̂₊) ← (∅, ∅)
2  while true
3     (Q̂, Q̂₋, Q̂₊) ← Clo(Q̂, Q̂₋, Q̂₊)
4     if (Q̂, Q̂₋, Q̂₊) satisfies the 𝒲 condition then
5        return "Run(S) ≠ ∅"
6     (Q̂, Q̂₋, Q̂₊) ← Red(Q̂, Q̂₋, Q̂₊)
7     if Q̂ = ∅ then
8        return "Run(S) = ∅"
9     else
10       Pick a kernel path π̂ in KerPath(Q̂, Q̂₋, Q̂₊)
11       if π̂ is feasible then
12          return "Run(S) ≠ ∅"
13       else  // π̂ is spurious
14          Pick a certified split (x̂, X̂, X̂₋, X̂₊)
15          if x̂ ∈ Q̂₋ then X̂₋ ← X̂
16          if x̂ ∈ Q̂₊ then X̂₊ ← X̂
17          Q̂ ← (Q̂ \ x̂) ∪ X̂
18          Q̂₋ ← (Q̂₋ \ x̂) ∪ X̂₋
19          Q̂₊ ← (Q̂₊ \ x̂) ∪ X̂₊
```

4.4.1 The PCegar algorithm

Intuitively, the PCegar algorithm is similar to the Cegar algorithm, except that certified approximations are used in place of cover abstractions. The initial certified pair is set to $(\emptyset, \emptyset)$ at line 1. Each iteration of the while loop starts with an application of closure to enlarge the certified pair. If the $\mathcal{W}$ condition of Proposition 4.2 holds (at line 4), then the algorithm returns "$Run(S) \neq \emptyset$". Otherwise, reduction is applied at line 6 to remove non-essential abstract states. If all abstract states have been eliminated, then "$Run(S) = \emptyset$" is returned. Otherwise, $\widehat{Q} \neq \emptyset$, which entails that there exists a kernel path (since reduction preserves kernel paths). The algorithm picks a kernel path at line 10. If this kernel path is feasible then $Run(S) \neq \emptyset$ is returned. Otherwise, a certified split is chosen at line 14, and is enlarged at lines 15–16. This enlargement step is crucial for correctness: without it, kernel paths may be lost after refinement, which could lead the algorithm to falsely return "$Run(S) = \emptyset$" at the next iteration. Next, refinement of the certified approximation is performed at lines 17–19, and the loop is iterated. Remark that the existence of a certified split at line 14 is guaranteed by the following observation: in a certified approximation, every spurious kernel path contains some abstract state $\widehat{x}$ with $|\widehat{x}| \geq 2$, which entails that the tuple $(\widehat{x}, \{\{q\} \mid q \in \widehat{x}\}, \emptyset, \emptyset)$ is a certified split.

In the PCegar algorithm, the closure, the test of the $\mathcal{W}$ condition, and the reduction are performed in an order that maximizes the possible gain of each operation. Observe that larger certified pairs lead to less kernel states. Therefore, we start by computing

the closure of the certified approximation, as this operation enlarges the certified pair, which benefits both the $\mathcal{W}$ test and the reduction. We then immediately test the $\mathcal{W}$ condition in order to shortcut the loop as soon as possible. If the $\mathcal{W}$ test fails, we apply reduction and then proceed along the same lines as Cegar. As shown by Proposition 4.4, any further combination of closure and reduction would be useless.

Before proving the correctness of PCegar, we first compare it with the classical Cegar algorithm. Let us consider an execution of Cegar, and try to inductively mimic it with PCegar (on the same input). To simplify the presentation, we assume that (a) each abstract run $\widehat{\pi}$ (picked by Cegar at line 2) is among the shortest ones (e.g., it was obtained by a breadth-first search of the cover abstraction), and (b) each refined abstract state $\widehat{x}$ (picked by Cegar at line 6) belongs to $\widehat{\pi}$. Suppose that both algorithms are at the beginning of their while loop, and that the abstract state space $\widehat{Q}$ of Cegar contains the abstract state space $\widehat{Q}^{\mathsf{P}}$ of PCegar. This is a reasonable assumption as both algorithms start with the same cover, and $\widehat{Q}^{\mathsf{P}}$ was obtained by mimicking Cegar, but with reductions. If Cegar exits the loop, i.e., there is no abstract run, then PCegar exits at line 8 since, according to Remark 4.1, applied on the certified cover abstraction $(\widehat{Q}, \widehat{Q}^{\mathsf{P}}_{-}, \widehat{Q}^{\mathsf{P}}_{+})$ there is no kernel path. Otherwise, Cegar picks some abstract run $\widehat{\pi}$, and checks its feasibility. This abstract run may not exist in the certified approximation maintained by PCegar at line 10. To mimic Cegar, it seems natural and fair to pick a kernel path $\widehat{\pi}^{\mathsf{P}}$ that is a *factor* of $\widehat{\pi}$ (i.e., a contiguous subsequence of $\widehat{\pi}$). If such a kernel path does not exist, then it follows from the correctness proof of PCegar (see below) that $\widehat{\pi}$ is spurious. Hence, Cegar refines its cover abstraction with some split $(\widehat{x}, \widehat{X})$, and iterates its loop. Observe that $\widehat{x} \notin \widehat{Q}^{\mathsf{P}}$, so we simply let PCegar ignore this iteration. Suppose now that PCegar picked a kernel path $\widehat{\pi}^{\mathsf{P}}$ that is a factor of $\widehat{\pi}$. If this kernel path is feasible, then PCegar returns at line 12. Remark that this may happen even though $\widehat{\pi}$ is spurious, in which case Cegar continues. If, on the contrary, $\widehat{\pi}^{\mathsf{P}}$ is spurious, then the abstract run $\widehat{\pi}$ is also necessarily spurious, and Cegar picks a split $(\widehat{x}, \widehat{X})$ to refine its cover abstraction. To mimic Cegar, it seems again natural and fair to pick a certified split that is an extension of $(\widehat{x}, \widehat{X})$. However, this is not always possible, since Cegar may choose $\widehat{x} \notin \widehat{Q}^{\mathsf{P}}$. In that case, we simply let PCegar ignore this iteration.

To conclude this comparison, the pruning performed by our PCegar algorithm has the following advantages:

- *Some useless refinements can be avoided*: Cegar may refine an abstract state that has been eliminated in PCegar. Put differently, each refinement performed by PCegar may remove, in a single step, several spurious abstract runs that would be considered by Cegar (see, for instance, the introduction's motivating example).
- *Counterexamples are shorter*: PCegar's counterexamples are factors of Cegar's counterexamples.
- *Counterexamples are more likely to be feasible*: Cegar may pick a counterexample that is spurious even though the corresponding counterexample of PCegar is feasible (or, worse, Cegar's spurious counterexample has been completely eliminated in PCegar).

- *Computational resources are reduced*: The abstract state space is reduced, which impacts the computation of the transition relation as well as the counterexample search. Moreover, counterexample feasibility analysis benefits from their shorter length.

In order to prove the correctness of PCegar, we will show that the certified approximations manipulated by the algorithm are "conservative" in the sense that they preserve non-emptiness of $Run(S)$, which is formalized as follows.

Definition 4.8. *A certified approximation* $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ *is* complete *if for each run* $q_0, \ldots, q_n$ *of* S *there exists* $0 \leq k \leq l \leq n$ *and a kernel path* $\widehat{q}_k, \ldots, \widehat{q}_l$ *such that:*

$$(k > 0 \Rightarrow \widehat{q}_k \in \widehat{Q}_-) \wedge (l < n \Rightarrow \widehat{q}_l \in \widehat{Q}_+) \wedge \bigwedge_{i=k}^{l} q_i \in \widehat{q}_i \tag{4.1}$$

Intuitively, in a complete certified approximation, each concrete run π of S is represented by some (feasible) kernel path $\widehat{\pi}$, in the sense that the concretization of $\widehat{\pi}$ contains a factor of π. Obviously, the same kernel path may represent several runs of S. Notice that every complete certified approximation satisfies the properties of Remark 4.1 (even if $\widehat{Q}$ is not a cover). The proof that PCegar is correct will use the following technical lemma.

Lemma 4.2. *Let* $(\widehat{Q}^1, \widehat{Q}^1_-, \widehat{Q}^1_+)$ *and* $(\widehat{Q}^2, \widehat{Q}^2_-, \widehat{Q}^2_+)$ *be two certified approximations satisfying* $(\bigcup \widehat{Q}^1) \subseteq (\bigcup \widehat{Q}^2)$, $(\bigcup \widehat{Q}^1_-) \subseteq (\bigcup \widehat{Q}^2_-)$, *and* $(\bigcup \widehat{Q}^1_+) \subseteq (\bigcup \widehat{Q}^2_+)$. *If* $(\widehat{Q}^1, \widehat{Q}^1_-, \widehat{Q}^1_+)$ *is complete then so is* $(\widehat{Q}^2, \widehat{Q}^2_-, \widehat{Q}^2_+)$.

Proof. Let $q_0, \ldots, q_n$ be a run of S. Since $(\widehat{Q}^1, \widehat{Q}^1_-, \widehat{Q}^1_+)$ is complete, there exists a kernel path $\widehat{q}^1_k, \ldots, \widehat{q}^1_l$ satisfying Equation 4.1, where $0 \leq k \leq l \leq n$. For every $k \leq i \leq l$, it holds that $q_i \in \widehat{q}^1_i$. By assumption, $(\bigcup \widehat{Q}^1) \subseteq (\bigcup \widehat{Q}^2)$. Therefore, there exists $\widehat{q}^2_k, \ldots, \widehat{q}^2_l$ in $\widehat{Q}^2$ such that $q_i \in \widehat{q}^2_i$ for all $k \leq i \leq l$. If $k > 0$ then $\widehat{q}^1_k \in \widehat{Q}^1_-$. Since $(\bigcup \widehat{Q}^1_-) \subseteq (\bigcup \widehat{Q}^2_-)$, it follows that $q_k \in (\bigcup \widehat{Q}^2_-)$, hence, we may choose $\widehat{q}^2_k$ such that $\widehat{q}^2_k \in \widehat{Q}^2_-$. Similarly, if $l < n$ then we may choose $\widehat{q}^2_l$ such that $\widehat{q}^2_l \in \widehat{Q}^2_+$. It is readily seen that $\widehat{q}^2_{i-1} \to \widehat{q}^2_i$ for every $k < i \leq l$. We have shown, so far, that there exists an abstract path $\widehat{q}^2_k, \ldots, \widehat{q}^2_l$ satisfying Equation 4.1. However, this abstract path is not necessarily a kernel path for $(\widehat{Q}^2, \widehat{Q}^2_-, \widehat{Q}^2_+)$. Let us define h and m, with $0 \leq h \leq m \leq n$, as follows:

$$\begin{aligned} h &= \max\Big(\{0\} \cup \{i \mid k \leq i \leq l \wedge \widehat{q}^2_i \in \widehat{Q}^2_-\}\Big) \\ m &= \min\Big(\{n\} \cup \{i \mid h \leq i \leq l \wedge \widehat{q}^2_i \in \widehat{Q}^2_+\}\Big) \end{aligned}$$

By construction, the abstract path $\widehat{q}^2_h, \ldots, \widehat{q}^2_m$ belongs to $KerPath(\widehat{Q}^2, \widehat{Q}^2_-, \widehat{Q}^2_+)$ and satisfies Equation 4.1. □

Proposition 4.3. *Given a cover* $\widehat{Q} \subseteq \mathcal{P}^+(Q)$, PCegar$(S, \widehat{Q})$ *is correct, and it terminates if* Q *is finite.*

Proof. It is readily seen that $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ remains a certified approximation inside the while loop (except, possibly, at lines 18–19). Therefore, if PCegar returns "$Run(S) \neq \emptyset$" (at line 5 or 12), we derive from Proposition 4.2 and Lemma 4.1 that this answer is correct. Assume now that PCegar returns "$Run(S) = \emptyset$" at line 8. To prove that this answer is correct, we show by induction that, at each iteration, the certified approximation $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ is complete at lines 3 and 7. Before the first iteration of the loop, since $\widehat{Q}$ is a cover of Q, each run of S is in the concretization of some abstract run, and it follows that $(\widehat{Q}, \emptyset, \emptyset)$ is complete. For the induction step, we observe that Lemma 4.2 entails that completeness is preserved under closure (line 3) and refinement (lines 15–19). Moreover, completeness is also preserved under reduction (line 6), since reduction obviously preserves kernel paths. This concludes the proof of the induction step, as well as the proof of correctness of PCegar. The proof of termination of PCegar for finite transition systems uses the same argument as for Cegar. □

4.5 On Optimality of Repeated Closures and Reductions

We now turn our attention to two aspects of our pruning method: the loss of kernel paths, and the benefit of different combination of the closure and reduction operations. We will start by the presentation of an example that highlights the impact of the closure operation, and then we discuss the impact of repeated reductions.

4.5.1 Loss of Kernel Paths by Reduction

In our PCegar algorithm, the closure operation (Line 3) is always performed before the application of the reduction operation (Line 6). As mentioned previously, this order maximizes the possible gain of each operation.

In particular, this order allows us to avoid the loss of kernel paths that could be eliminated by the reduction operation. These kernel paths could satisfy the $\mathcal{W}$ condition, and, therefore, let PCegar conclude earlier. For instance, the example given in Figure 4.2 illustrates the loss of a kernel path by reduction when $Ker(Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+))$ is not equal to $Ker(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$. Indeed observe that $Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ and $Red(Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+))$ contain the witness path $\widehat{A} \to \widehat{B}$ whereas $Red(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ does not. On the other hand, extending the sets $\widehat{Q}_-$, and $\widehat{Q}_+$ by closure (or any certified pair inference method) may weaken the impact of the reduction. Indeed, on the same example, the certified approximation $Red(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ contains less states that $Red(Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+))$.

4.5.2 Closure and Reduction Ordering

Recall that closure and reduction are operations that improve the "quality" of certified approximations: they lead to larger certified pairs and smaller abstract state spaces. These operations are idempotent. But one may be tempted to alternate closure and reduction several times instead of just once (lines 3–6) at each iteration. The following proposition shows that the certified approximation $Red(Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+))$ obtained at

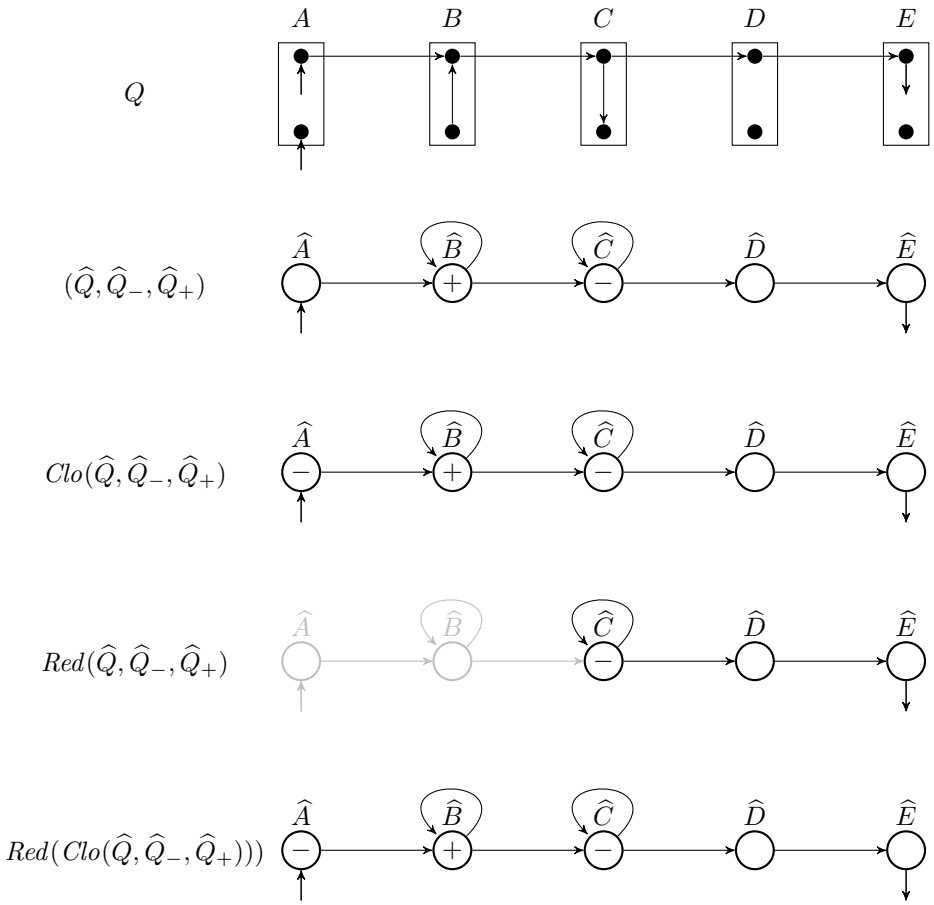


Figure 4.2: An example that illustrate the loss of kernel paths by reduction.

line 6 of PCegar is optimal in the sense that applying reduction and closure multiple times and in any order would produce the same result. Note that the loss of kernel paths by reduction (before closure) jeopardize the correction of the algorithm.

Proposition 4.4. *Let $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ be a certified approximation. If $Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$ does not satisfy the $\mathcal{W}$ condition, then it holds that:*

$$\begin{aligned} Red(Clo(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)) &= Clo(Red(Clo((\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+))) \\ &= Red(Clo(Red((\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+))) \end{aligned}$$

The remaining of this section is dedicated to the proof of this proposition.

Consider a certified approximation $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$. We will use the following notations:

$$\begin{array}{rcl}
(\widehat{Q}^c, \widehat{Q}^c_-, \widehat{Q}^c_+) & = & \mathit{Clo}(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+) \\
(\widehat{Q}^r, \widehat{Q}^r_-, \widehat{Q}^r_+) & = & \mathit{Red}(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+) \\
(\widehat{Q}^{cr}, \widehat{Q}^{cr}_-, \widehat{Q}^{cr}_+) & = & \mathit{Clo}(\mathit{Red}(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)) \\
(\widehat{Q}^{rc}, \widehat{Q}^{rc}_-, \widehat{Q}^{rc}_+) & = & \mathit{Red}(\mathit{Clo}(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)) \\
(\widehat{Q}^{crc}, \widehat{Q}^{crc}_-, \widehat{Q}^{crc}_+) & = & \mathit{Clo}(\mathit{Red}(\mathit{Clo}(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+))) \\
(\widehat{Q}^{rcr}, \widehat{Q}^{rcr}_-, \widehat{Q}^{rcr}_+) & = & \mathit{Red}(\mathit{Clo}(\mathit{Red}(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)))
\end{array}$$

We first prove that $(\widehat{Q}^{crc}, \widehat{Q}^{crc}_-, \widehat{Q}^{crc}_+) = (\widehat{Q}^{rc}, \widehat{Q}^{rc}_-, \widehat{Q}^{rc}_+)$. It follows from the definition of closure (Definition 4.6) that $\widehat{Q}^{crc} = \widehat{Q}^{rc}$, $\widehat{Q}^{crc}_- \supseteq \widehat{Q}^{rc}_-$ and $\widehat{Q}^{crc}_+ \supseteq \widehat{Q}^{rc}_+$. Moreover, every $\widehat{q} \in \widehat{Q}^{crc}_-$ is reachable in $S[\widehat{Q}^{rc}]$ from $\widehat{Q}^{rc}_- \cup (\widehat{I}^{rc} \cap \mathcal{P}(I))$ by must $\xrightarrow{-}$ transitions, which entails that $\widehat{q} \in \widehat{Q}^c_-$, hence, $\widehat{q} \in \widehat{Q}^{rc}_-$. We get that $\widehat{Q}^{crc}_- \subseteq \widehat{Q}^{rc}_-$, and, similarly, $\widehat{Q}^{crc}_+ \subseteq \widehat{Q}^{rc}_+$.

Let us now assume that $(\widehat{Q}^c, \widehat{Q}^c_-, \widehat{Q}^c_+)$ does not satisfy the $\mathcal{W}$ condition of Proposition 4.2. The proof that $(\widehat{Q}^{rcr}, \widehat{Q}^{rcr}_-, \widehat{Q}^{rcr}_+) = (\widehat{Q}^{rc}, \widehat{Q}^{rc}_-, \widehat{Q}^{rc}_+)$ will follow from the following technical lemmas.

Lemma 4.3. *For every $\widehat{q} \in \widehat{Q}^c_-$, there is a path $\widehat{p}_0, \ldots, \widehat{p}_m$ in $S[\widehat{Q}]$, with $\widehat{p}_0 \in (\widehat{Q}_- \cup (\widehat{I} \cap \mathcal{P}(I)))$ and $\widehat{p}_m = \widehat{q}$, such that, for all $0 < i \leq m$, we have $\widehat{p}_i \notin (\widehat{Q}_- \cup \widehat{Q}_+)$ and $\widehat{p}_{i-1} \xrightarrow{-} \widehat{p}_i$.*

Proof. If $\widehat{q} \in \widehat{Q}_-$ then we simply take the path consisting of $\widehat{q}$. Suppose now that $\widehat{q} \in (\widehat{Q}^c_- \setminus \widehat{Q}_-)$. We derive from the definition of closure (Definition 4.6) that there is a path $\widehat{p}_0, \ldots, \widehat{p}_m$ in $S[\widehat{Q}]$, with $\widehat{p}_0 \in (\widehat{Q}_- \cup (\widehat{I} \cap \mathcal{P}(I)))$ and $\widehat{p}_m = \widehat{q}$, such that $\widehat{p}_{i-1} \xrightarrow{-} \widehat{p}_i$ for all $0 < i \leq m$. Furthermore, we may obviously assume w.l.o.g. that $\widehat{p}_i \notin \widehat{Q}_-$ for all $0 < i \leq m$. Observe that $\widehat{p}_i \in \widehat{Q}^c_-$ for all $0 \leq i \leq m$. It follows that $\widehat{p}_i \notin \widehat{Q}_+$ for all $0 \leq i \leq m$, as otherwise $(\widehat{Q}^c, \widehat{Q}^c_-, \widehat{Q}^c_+)$ would satisfy the $\mathcal{W}$ condition. We get that $\widehat{p}_i \notin (\widehat{Q}_- \cup \widehat{Q}_+)$ for all $0 < i \leq m$. □

Lemma 4.4. *For every $\widehat{q} \in \widehat{Q}^c_+$, there is a path $\widehat{p}_0, \ldots, \widehat{p}_m$ in $S[\widehat{Q}]$, with $\widehat{p}_0 = \widehat{q}$ and $\widehat{p}_m \in (\widehat{Q}_+ \cup (\widehat{F} \cap \mathcal{P}(F)))$, such that, for all $0 \leq i < m$, we have $\widehat{p}_i \notin (\widehat{Q}_- \cup \widehat{Q}_+)$ and $\widehat{p}_i \xrightarrow{+} \widehat{p}_{i+1}$.*

Proof. Similar to the proof of Lemma 4.3. □

Lemma 4.5. *Every kernel path of $(\widehat{Q}^c, \widehat{Q}^c_-, \widehat{Q}^c_+)$ is a factor of some kernel path of $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$.*

Proof. Let $\widehat{\pi} = \widehat{q}_0, \ldots, \widehat{q}_n$ be a kernel path of $(\widehat{Q}^c, \widehat{Q}^c_-, \widehat{Q}^c_+)$. Since $\widehat{Q}_- \subseteq \widehat{Q}^c_-$ and $\widehat{Q}_+ \subseteq \widehat{Q}^c_+$, it holds that $\widehat{q}_i \notin (\widehat{Q}_- \cup \widehat{Q}_+)$ for all $0 < i < n$. We show that there exists two paths $\widehat{\mu}, \widehat{\nu}$ such that $\widehat{\mu} \cdot \widehat{\pi} \cdot \widehat{\nu}$ is a kernel path in $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$. If $\widehat{q}_0 \in (\widehat{I} \cup \widehat{Q}_-)$ then we set $\widehat{\mu}$ to the empty path. Otherwise, we set $\widehat{\mu}$ to the path obtained with Lemma 4.3. We proceed symmetrically for $\widehat{\nu}$ (with Lemma 4.4), and we obtain that $\widehat{\mu} \cdot \widehat{\pi} \cdot \widehat{\nu}$ is a kernel path in $(\widehat{Q}, \widehat{Q}_-, \widehat{Q}_+)$. □

Lemma 4.6. *It holds that* $\widehat{Q}^{cr}_{-} = \widehat{Q}^{r} \cap \widehat{Q}^{c}_{-}$ *and* $\widehat{Q}^{cr}_{+} = \widehat{Q}^{r} \cap \widehat{Q}^{c}_{+}$.

Proof. We only show the equality $\widehat{Q}^{cr}_{-} = \widehat{Q}^{r} \cap \widehat{Q}^{c}_{-}$, the proof that $\widehat{Q}^{cr}_{+} = \widehat{Q}^{r} \cap \widehat{Q}^{c}_{+}$ is similar. The inclusion $\widehat{Q}^{cr}_{-} \subseteq \widehat{Q}^{r} \cap \widehat{Q}^{c}_{-}$ directly follows from the definitions of reduction (Definition 4.5) and closure (Definition 4.6). Let us prove the reverse inclusion. Let $\widehat{q} \in \widehat{Q}^{r} \cap \widehat{Q}^{c}_{-}$. If $\widehat{q} \in \widehat{Q}_{-}$ then we get that $\widehat{q} \in \widehat{Q}^{r}_{-}$, which entails that $\widehat{q} \in \widehat{Q}^{cr}_{-}$. Assume now that $\widehat{q} \notin \widehat{Q}_{-}$. Let $\widehat{\pi} = \widehat{p}_0, \ldots, \widehat{p}_m$ denote the path obtained with Lemma 4.3. Since $\widehat{q} \in \widehat{Q}^{r}$, there exists a kernel path $\widehat{\rho} = \widehat{r}_0, \ldots, \widehat{r}_n$ in $(\widehat{Q}, \widehat{Q}_{-}, \widehat{Q}_{+})$ such that $\widehat{r}_k = \widehat{q}$ for some $0 \leq k \leq n$. We have $\widehat{r}_i \notin (\widehat{Q}_{-} \cup \widehat{Q}_{+})$ for all $k < i < n$. It follows that the path $\widehat{\pi} \cdot (\widehat{r}_k, \ldots, \widehat{r}_n)$ is also a kernel path, which entails that $\widehat{p}_i \in \widehat{Q}^{r}$ for all $0 \leq i \leq m$. We conclude that $\widehat{\pi}$ is a path in $S[\widehat{Q}^{r}]$, hence, $\widehat{q} \in \widehat{Q}^{cr}_{-}$. □

Lemma 4.7. *It holds that* $\widehat{Q}^{rcr} = \widehat{Q}^{rc}$.

Proof. We show that $(\widehat{Q}^{cr}, \widehat{Q}^{cr}_{-}, \widehat{Q}^{cr}_{+})$ and $(\widehat{Q}^{c}, \widehat{Q}^{c}_{-}, \widehat{Q}^{c}_{+})$ have the same kernel paths. Let $\widehat{q}_0, \ldots, \widehat{q}_n$ be a kernel path in $(\widehat{Q}^{cr}, \widehat{Q}^{cr}_{-}, \widehat{Q}^{cr}_{+})$. Note that $\widehat{Q}^{cr} = \widehat{Q}^{r}$. Hence, $\widehat{q}_i \in \widehat{Q}^{r}$ for all $0 \leq i \leq n$. It follows from Lemma 4.6 that $\widehat{q}_0, \ldots, \widehat{q}_n$ is a kernel path in $(\widehat{Q}^{c}, \widehat{Q}^{c}_{-}, \widehat{Q}^{c}_{+})$.

Conversely, let $\widehat{q}_0, \ldots, \widehat{q}_n$ be a kernel path in $(\widehat{Q}^{c}, \widehat{Q}^{c}_{-}, \widehat{Q}^{c}_{+})$. We derive from Lemma 4.5 that $\widehat{q}_i \in \widehat{Q}^{r}$ for all $0 \leq i \leq n$. Note that $\widehat{Q}^{cr} = \widehat{Q}^{r}$. It follows from Lemma 4.6 that $\widehat{q}_0, \ldots, \widehat{q}_n$ is a kernel path in $(\widehat{Q}^{cr}, \widehat{Q}^{cr}_{-}, \widehat{Q}^{cr}_{+})$. □

To complete the proof of the proposition, it remains to show that $\widehat{Q}^{rcr}_{-} = \widehat{Q}^{rc}_{-}$ and $\widehat{Q}^{rcr}_{+} = \widehat{Q}^{rc}_{+}$. Recall from Lemma 4.7 that $\widehat{Q}^{rcr} = \widehat{Q}^{rc}$. Let $\widehat{q} \in \widehat{Q}^{rcr}$. Since $\widehat{Q}^{rcr} \subseteq \widehat{Q}^{cr} = \widehat{Q}^{r}$, we get that $\widehat{q} \in \widehat{Q}^{r}$. We obtain that:

$$\begin{array}{lcll} \widehat{q} \in \widehat{Q}^{rcr}_{-} & \Leftrightarrow & \widehat{q} \in \widehat{Q}^{cr}_{-} & (\widehat{q} \in \widehat{Q}^{rcr}) \\ & \Leftrightarrow & \widehat{q} \in \widehat{Q}^{c}_{-} & (\text{Lemma 4.6}) \\ & \Leftrightarrow & \widehat{q} \in \widehat{Q}^{rc}_{-} & (\widehat{q} \in \widehat{Q}^{rc}) \end{array}$$

This entails that $\widehat{Q}^{rcr}_{-} = \widehat{Q}^{rc}_{-}$. The proof that $\widehat{Q}^{rcr}_{+} = \widehat{Q}^{rc}_{+}$ is similar.

4.6 Implementation and Experimentation

We now briefly present our implementation of the PCegar algorithm, and compare it with Cegar on a suite of finite-state systems. In Chapter 6 we will present with more details our implementation of the PCegar algorithm, and analyze the behavior of both algorithm on some benchmark models.

We have implemented Cegar and PCegar as an extension of the Mec 5 model checker [GV04, Mec10]. Mec 5 manages finite relations with BDDs. The search for abstract counterexamples is performed in a breadth-first manner.

In general the computation of the closure operation may require "expensive" operations due to the must transition relations. In our experiments, the extra pruning obtained with the closure did not compensate these extra BDD computations. Therefore, we disabled the closure operation for the experimentations presented below.

For both algorithms, refinement of abstract transitions is done locally: given a split $(\widehat{x}, \widehat{X})$, we decide for each $\widehat{y} \in \widehat{X}$ and $\widehat{q} \to \widehat{x}$ (resp. $\widehat{x} \to \widehat{q}$) whether $\widehat{q} \to \widehat{y}$ (resp. $\widehat{y} \to \widehat{q}$). For the PCegar algorithm, another optimization is allowed by the use of kernel paths: incoming transitions to states in $\widehat{Q}_-$ can be discarded as they will never be part of a kernel path, and likewise outgoing transitions from $\widehat{Q}_+$ states can be discarded. This allows us to avoid useless computations of abstract transitions when refinement is performed.

For feasibility checking, our prototype analyses an abstract counterexample by computing the iterated concrete *post* or *pre* along it. The main refinement heuristics implemented in our tool are `Post` and `Pre`. The heuristic `Post` is the adaptation of the refinement proposed in [CGJ+03] as discussed in Section 4.3. The `Pre` heuristic is the dual of `Post`.

For a meaningful comparison of Cegar and PCegar, as discussed in Section 4.4, we forced PCegar to pick a kernel path that is a factor of an abstract run that would be picked by Cegar. Therefore, the kernel path picked by PCegar is not necessarily among the shortest ones. However, we will see that, on many examples, PCegar is still capable to conclude with less loop iterations. Notice that the actual splits computed by PCegar may be different from those of Cegar, as the iterated *post* (or *pre*) computation along the counterexample starts from a different abstract state. We also applied the classical abstraction reduction to Cegar, which removes abstract states that are not reachable or not co-reachable. Observe that this minimal reduction is a particular instance of PCegar (without closure) where the sets $\widehat{Q}_-$ and $\widehat{Q}_+$ remain empty during its execution.

		Time (s)		*Memory (MiB)*		*Loops*		*Average* $\lvert\widehat{\pi}\rvert$		*C.E. Analysis*		*Average* $\lvert\widehat{Q}\rvert$		*Trans. Analysis*	
		C	PC	C	PC	C	PC	C	PC	C	PC	C	PC	C	PC
Burns 2	Post	13	3	165	80	465	249	21	5	8784	762	258	54	102706	10876
	Pre	7	1	65	57	476	382	20	5	8551	1221	254	68	31538	11402
Burns 3	Post	247	122	2989	1265	2857	1076	21	4	57913	3261	1540	218	1228572	55262
	Pre	×	25	×	188	×	5182	×	6	×	16251	×	1028	×	330342
Lamport 2	Post	4	3	68	54	213	137	16	7	2702	461	147	51	3696	2252
	Pre	3	2	50	52	213	196	21	10	2759	732	70	63	3330	2996
Lamport 3	Post	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Pre	1366	1406	3944	3975	8563	8385	53	30	185520	29830	724	696	214426	196826
Lift 5	Post	20	9	87	83	124	124	13	3	1585	372	68	15	6432	2958
	Pre	1788	8	3358	79	2777	92	736	3	1243076	279	1394	13	259254	2362
Lift 6	Post	95	46	163	164	182	182	14	3	2605	546	98	18	10872	4652
	Pre	×	24	×	126	×	135	×	3	×	409	×	15	×	4088
Lift 7	Post	400	208	303	288	250	250	16	3	3950	750	133	22	16694	6670
	Pre	×	82	×	286	×	186	×	3	×	563	×	18	×	6508
Lift 8	Post	1540	857	540	540	328	328	17	3	5662	984	173	25	24042	9004
	Pre	×	290	×	538	×	245	×	3	×	741	×	21	×	9750
Peterson 2	Post	0	0	8	8	19	19	8	6	104	70	27	20	254	230
	Pre	0	0	8	8	19	17	14	6	176	63	26	21	256	206
Peterson 3	Post	7	7	131	119	416	405	14	5	4916	1291	264	94	14668	11134
	Pre	10	5	120	81	416	404	108	16	24859	1432	257	88	16112	10700

Table 4.1: Comparison of Cegar (C) and PCegar (PC) on a suite of finite state examples. These experiments have been performed on an Intel Xeon 2.33 GHz. Computation time in seconds, memory usage in mebibytes, number of CEGAR loop iterations, average size of abstract state space, average length of abstract counterexample, total number of *post*/*pre* computations for (a) abstract counterexample analysis, and (b) abstract transition refinement. The sign × means that the verification did not terminate within 1800 seconds or required more than 5 GiB of memory.

Experimentations have been performed using two types of models [PLA10]: classical mutual exclusion algorithms on which we checked the mutual exclusion property, and a classical lift model, on which we checked that floor doors can be opened only if the elevator is there. The models are parameterized with the number of processes in the case of mutual exclusion algorithms, and with the number of floors for the lift. We use, as initial abstraction, the partition abstraction induced by the control locations of each process.

Table 4.1 presents the results obtained with our implementation. As expected, PCegar outperforms (or is equivalent in some cases) Cegar on loop iterations, counterexample analysis, and transition refinements. The loop iterations criterion shows a great benefit on many cases, the gain ranges from a few iterations to a factor of 42 times less loop iterations. Yet on some examples, PCegar requires as many loop iteration as Cegar, but this comes from our choice that forces PCegar to select counterexamples that would be picked by Cegar. Pruning reduces dramatically the average size of the abstractions: the state space reduction goes up to two orders of magnitude. This has a direct impact on the cost of abstract transition refinement (total number of *post*/*pre* operations) which exhibits similar gains. Likewise, the use of kernel paths as counterexamples allows PCegar to pick counterexamples that are, in average, much smaller (up to two orders of magnitude) than those selected by Cegar. This translates into a similar improvement for counterexample analysis (number of *post*/*pre* operations).

We also report on time and memory requirements of both algorithms. Regarding computation times, PCegar outperforms Cegar which is quite natural due to the avoided operations. Comparing memory requirements is less significant due to the BDD manager implemented in Mec 5, that uses a lazy garbage collector. Nevertheless, we observe that PCegar uses, in general, less memory than Cegar. Again, this comes from the abstract state pruning that discards useless BDDs.

4.7 Concluding Remarks

In this chapter, we have presented an improvement of the classical CEGAR paradigm with abstract state pruning. Our goal was to accelerate the CEGAR loop in a generic way that takes advantage of the computation performed by a CEGAR model checker. To this end we have presented certified pairs. This allows to introduce pruning abstractions, and the use of certified approximations. This pruning not only reduces the computational resources to maintain and explore the abstraction, but also leads to fewer refinement steps and focuses the algorithm on abstract counterexamples that are shorter and more likely to be feasible. The experimentations that we performed with a BDD-based model checker, demonstrated the expected gain over the standard Cegar algorithm. In Chapter 6.3 we will detail the results, and present the benefits of pruning during the execution of a CEGAR loop.

We have experimented on finite-state models, but our abstraction pruning technique also applies to infinite-state systems, and we expect improvements for these systems too. Extending our certified pair inference method at the syntax level, is a challenging and

promising direction. This will widen the scope of our PCegar algorithm to a larger variety of model checkers.

Chapter 5

Compositional CEGAR

Introduction

Transition systems are rarely given explicitly. Instead, they are often presented as some form of parallel composition of basic transition systems. Modular presentation takes this idea one step further, it allows to apply parallel compositions in a hierarchical manner. The semantics of such a hierarchical system is just a standard transition system. It is easy to see though that the semantics can be exponentially bigger than its hierarchical representation. In this section we investigate methods of doing CEGAR abstractions of transition systems presented in a hierarchical way. The objective is to avoid calculating semantics of the hierarchical system, and moreover to take advantage of the presentation of a system in the form of modules in order to find useful abstractions quicker.

Hierarchical representation is based on parallel composition. Here we consider synchronous product of transition systems with synchronization vectors [AN82]. In the simplest version of the synchronous product, the product of two systems can do an action if both components can do it. In a more elaborate synchronous product with communication vectors an action of one system can be synchronized with another action of the other system if there is a synchronization vector containing the two actions. This extension allows for a more flexible product, and it is very useful especially in hierarchical representation (cf. the example of stack in Chapter 2.1.2).

Priorities are another powerful feature we consider in this section. A priority relation is a partial order on actions. If both actions a and b are possible from a state, but b has higher priority than a then it is b that will be executed. In other words action a will be blocked. Seen like this, priorities are quite trivial. Their power shows up when used together with parallel composition. Suppose that both a and b demand some synchronization to be executed: say a needs to be synchronized with c, and b with d, respectively. If the other component proposes c but not d then the synchronization of a and c will be executed even though b has higher priority. In contrast if the other

system proposes both c and d then only synchronization of b and d will be executed. So priorities have also somehow branching-time flavor: they allow to detect that some action is not possible. Priorities are very useful for modeling (with their help we can simplify the model description, for more details, see Chapter 2.1.2), but they pose real theoretical challenges in particular for CEGAR method. This can be attributed in part to their contravariant nature: adding more transitions in one component can eliminate some transitions in the other.

The objective of this chapter is to extend CEGAR approach to hierarchical systems. The simplest method is to calculate the semantics of a hierarchical system and then to apply any of standard CEGAR algorithms. For the reasons of size explosion this may not always be feasible. Instead we study methods of applying CEGAR approach without calculating the semantics of the hierarchical system. We go one step further and suppose that the abstract system itself should be hierarchical: it should reflect the hierarchy of the analyzed system. This gives an interesting situation when each component is abstracted separately, so the abstract system is represented in a succinct way too. The first obstacle in this approach is that in general a hierarchical composition of abstractions may not be an abstraction. We show that the notion of cover abstraction adapts well to hierarchical setting without priorities. We show moreover that in this case it is easy to verify if an abstract path is feasible: it is enough to look at the projection of this path into each of the components.

In the presence of priorities the situation is much more complicated. Due to above mentioned contravariant nature of priorities, it is not even clear how to guarantee that a hierarchical composition of abstractions is an abstraction. To circumvent this problem, we introduce a concept of neat cover abstraction. We show that it allows to recover most of the properties of the setting without priorities.

Finally, we will also discuss symbolic representations of hierarchical transition systems. We will show how to use AltaRica formalism to represent hierarchical transition systems, construct initial abstractions, and do abstraction refinement.

5.1 Hierarchical Transition Systems

A hierarchical transition system is a tree of transition systems together with synchronization vectors telling what synchronizations between those systems are possible. We have chosen to single out the hierarchical structure with a help of a notion of hierarchical schema. This way, later we will be able to consider abstractions that are instances of the same schema as the original system. As it will turn out, in the presence of priorities the semantics of a hierarchical transition system is quite subtle. For the reasons of compositionality, priorities need to be resolved at each level of the hierarchy. We will show that it is not a priory possible to remove priorities, or to delay them by moving them up in the hierarchy.

5.1.1 Hierarchical representation and its semantics

Hierarchical Schema

A hierarchical schema is a tree labeled with signatures of transition systems. A tree t is a prefix closed subset of $\mathbb{N}^*$ satisfying the property that if vi is in t then vj is in t for all $j < i$. The root of the tree is the empty sequence denoted by λ. A sequence vi is a *successor* of a sequence v. Two nodes vi, vj, namely the nodes that differ only in the last element, are called *siblings*. The order on siblings is inherited from that on $\mathbb{N}$. We will consider only finite trees.

A *hierarchical schema* is a tuple $\mathcal{S} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}\rangle$ where for every node v of the tree t:

- Σ_v is a finite alphabet of events,
- $\preccurlyeq_v$ is a partial order on Σ_v,
- $\delta_v \subseteq \Sigma_v \times \Sigma_{v0} \times \cdots \times \Sigma_{vk}$ is a set of synchronization vectors; here $v0, \ldots, vk$ are all the successors of v in t.

We call the partial order $\preccurlyeq_v$ the *priority relation* of v, and if $\preccurlyeq_v$ is the equality for all v in t then $\mathcal{S}$ is called *priority free*. We use the notion $\prec$ for the strict version of $\preccurlyeq$.

In Figure 5.1(a) we have depicted a hierarchical schema. The nodes of the tree t are represented as follows: the name of the node, and below in a box we have from left to right: an alphabet, a priority relation, and synchronization vectors of the node. The priority relation is supposed to be the smallest partial order generated by the displayed pairs of actions. The node λ for instance defines an alphabet made of a single event a, the trivial priority relation, and the synchronization vector saying that the a event synchronizes with ε event of its 0 successor and b even of its 1 successor. In 1 node we can see a nontrivial priority relation $b \prec a$.

Hierarchical Transition System

Hierarchical schema gives us a skeleton of a hierarchical transition system. The later is given by simply providing for each node of a hierarchy tree a transition system over an appropriate alphabet. Formally *hierarchical transition system* is a tuple

$$\mathcal{H} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}, \{S_v\}_{v\in t}\rangle$$

where $\langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}\rangle$ is a hierarchical schema, and $S_v = \langle Q_v, \Sigma_v, \rightarrow_v, I_v, F_v\rangle$ is a transition system over the alphabet of actions Σ_v; i.e., the alphabet given by the schema

Graphically, we use two representations of hierarchical transition systems. The first representation is a hierarchical schema as in Figure 5.1(a) together with transition systems, and a mapping that define the transition system associated to a node (see Figure 5.5).

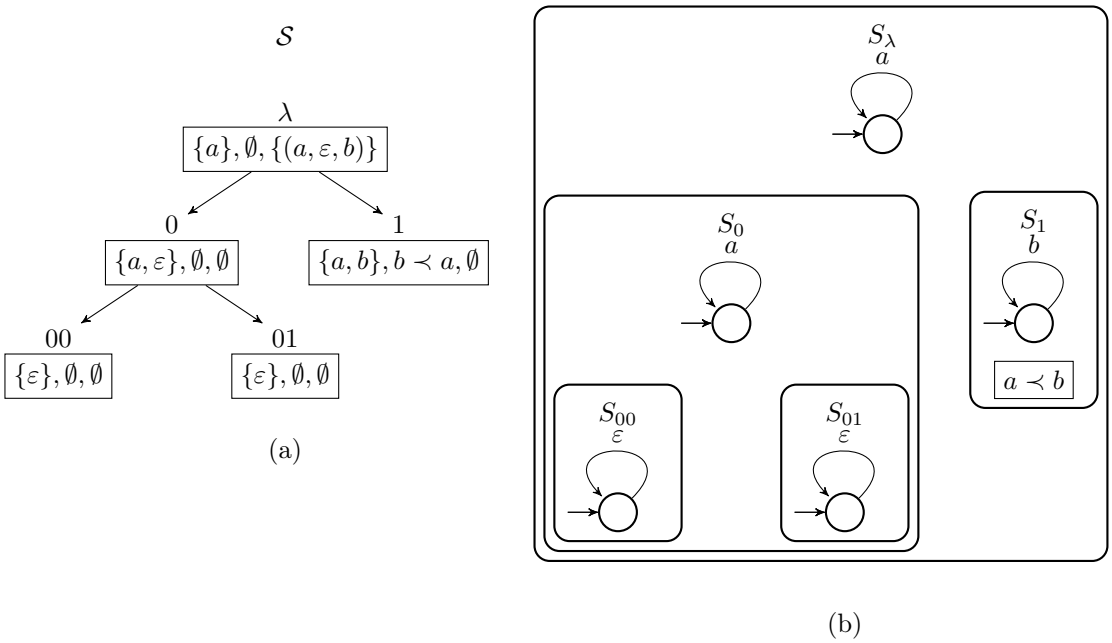


Figure 5.1: A hierarchical transition system, (a) the hierarchical schema $\mathcal{S}$, (b) an associated hierarchical transition system.

The second is inspired of the AltaRica nodes (see Chapter 2.2.5), and an example is given in Figure 5.1(b). This hierarchical transition system is associated to the hierarchical schema $\mathcal{S}$ Figure 5.1(a). Observe that the tree structure in the hierarchical transition system is depicted using englobing boxes containing its transition system, and its sub hierarchical schemata. In the upcoming examples using this representation, we will only represent graphically the priority relation of each node when it has one: for instance the node S_1 defines the priority $a \prec b$.

Semantics of a Hierarchical Transition System

The semantics of a hierarchical system is a standard transition system obtained as the synchronized product of its components. The meaning is a part of the complete synchronous product as synchronizations are limited by synchronization vectors. Moreover there are priorities that come into play forbidding some actions to happen. The interplay of these phenomena makes the semantics of a hierarchical system quite complex.

By induction on the height of a node v in the hierarchy we define the semantics of the hierarchical system determined by the subtree $t{\downarrow_v}$ of t:

$$S_v^\flat = \langle Q_v^\flat, \Sigma_v^\flat, \hookrightarrow_v, I_v^\flat, F_v^\flat \rangle.$$

If v is a leaf then $S_v^\flat$ is just S_v, the system assigned to v, with some actions removed due to the synchronization vectors and priorities. This means that $\Sigma_v^\flat = \delta_v$, and instead of

$\rightarrow_v$ we take $\hookrightarrow_v$ defined by $q \overset{e}{\hookrightarrow}_v q'$ if:

- $q \overset{e}{\rightarrow}_v q'$ and for no $e'' \neq e$ such that $e \preccurlyeq_v e''$ we have $q \overset{e''}{\rightarrow}_v q''$.

Suppose now that v is an internal node of the hierarchy and suppose that we know the semantics of hierarchical systems determined by the successors $v0, \ldots, vk$ of of v in t. The *semantic* $S_v^\flat$ is a transition system over the set of states

- $Q_v^\flat = Q_v \times Q_{v0}^\flat \times \cdots \times Q_{vk}^\flat$,

Observe that an element of $Q_v^\flat$ can be seen as a function from the subtree rooted in v, namely $t{\downarrow_v}$, to sets of states of respective components. So we can write $q(\lambda)$ for the state labeling the root of this subtree, that is a state of Q_v. Given a node u of the tree we can write $q{\downarrow_u}$ for the restriction of q to $t{\downarrow_u}$. We write $q(u)$ for the state of the component u, or to say it differently, for the state in the root of $q{\downarrow_u}$.

The set of initial states is easy to define:

- $I_v^\flat = I_v \times I_{v0}^\flat \times \cdots \times I_{vk}^\flat$,

it is just a set of tuples consisting of initial states only.

The set of final states is a bit more difficult to describe. This comes from our view of final states as being error states. So the whole system is in an error state if one of its components is in an error state:

- $F_v^\flat = \{q \in Q_v^\flat : \text{exists } u \text{ in } t{\downarrow_v} \text{ such that } q(u) \in F_u\}$.

The alphabet $\Sigma_v^\flat$ will be also a product: $\Sigma_v^\flat \subseteq \Sigma_v \times \Sigma_{v0}^\flat \times \cdots \times \Sigma_{vk}^\flat$. But this time we take into account allowed synchronization vectors:

- $\Sigma_v^\flat = \{e : (e(v), e(v0), \ldots, e(vk)) \in \delta_v\}$

In order to define the transition relation $\hookrightarrow_v$ we will first define auxiliary relation $\rightsquigarrow_v$. This relation will ignore priorities in v but it will take into account priorities in subcomponents of v. We say that $q \overset{e}{\rightsquigarrow}_v q'$ if

- $q(\lambda) \overset{e(\lambda)}{\rightarrow}_v q'(\lambda)$, and
- $q{\downarrow_i} \overset{e{\downarrow_i}}{\hookrightarrow}_{v_i} q'{\downarrow_i}$.

The relation $\hookrightarrow_v$ takes into account the priorities in v. We say that $q \overset{e}{\hookrightarrow}_v q'$ if

- $q \overset{e}{\rightsquigarrow}_v q'$ and
- for no e'' such that $e(\lambda) \prec_v e''(\lambda)$ we have $q \overset{e''}{\rightsquigarrow}_v q''$.

In other words a transition on e is possible, if it is possible without looking at the priorities in v, and moreover no action with higher priority is possible.

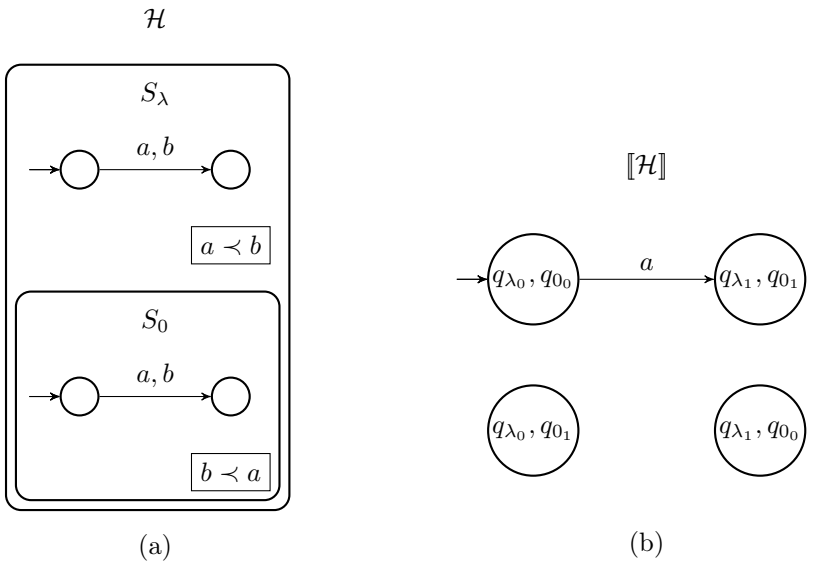


Figure 5.2: (a) a hierarchical transition system $\mathcal{H}$, (b) the semantic of $\mathcal{H}$.

Definition 5.1. *Let $\mathcal{H}$ be a hierarchical transition system, and let $S_v^\flat$ for very node v of the hierarchy tree be the transition system as defined above. The semantics of $\mathcal{H}$, denoted $[\![\mathcal{H}]\!]$, is $S_\lambda^\flat$ where λ is the root of the hierarchy tree.*

Consider the hierarchical transition system $\mathcal{H}$ given in Figure 5.2(a). $\mathcal{H}$ is composed of two nodes: λ and 0, the 0 node has the priority $b \prec a$. The node λ has the priority $a \prec b$, and the synchronization vectors of the λ node are: $\{(a,a),(b,b)\}$. The transition system S_λ is defined as follows: $S_\lambda = \langle\{q_0,q_1\},\{\varepsilon,a,b\},\rightarrow_\lambda,\{q_0\},\emptyset\rangle$, the transition relation is $\rightarrow_\lambda = \{(q_0,\varepsilon,q_0);(q_1,\varepsilon,q_1);(q_0,a,q_1);(q_0,b,q_1)\}$. The transition system S_0 is defined as follows: $S_0 = \langle\{q_0,q_1\},\{\varepsilon,a,b\},\rightarrow_0,\{q_0\},\emptyset\rangle$. Let us now compute the semantic of $\mathcal{H}$. By definition we have $[\![\mathcal{H}]\!] = \langle Q,\Sigma,\hookrightarrow,I,F\rangle$ where: $Q = Q_\lambda \times Q_0^\flat$, $I = I_\lambda \times I_0^\flat$, $F = F_\lambda \times F_0^\flat$, and the transition relation $\hookrightarrow$ is defined as:

- $\hookrightarrow = \{q \overset{e}{\rightsquigarrow}_\lambda q' \mid e' \in out(q) \Rightarrow e' \preccurlyeq e\}$

We therefore need the relation $\rightsquigarrow_\lambda$ that is defined as:

- $\rightsquigarrow_\lambda = \{q(\lambda) \overset{e}{\rightarrow}_0 q'(\lambda) \mid q{\downarrow_0} \overset{e{\downarrow_0}}{\hookrightarrow}_0 q{\downarrow'_0}\}$

Observe that we need the transition relation $\hookrightarrow_0$, and therefore the relation $\rightsquigarrow_0$. Finally, in order to get the $\hookrightarrow_\lambda$ relation we solve the following relation in the given order:

1. $\rightsquigarrow_0 = \{(q_0,a,q_1);(q_0,b,q_1)\}$,
2. $\hookrightarrow_0 = \{(q_0,a,q_1)\}$,

3. $\rightsquigarrow_\lambda = \{((q_{\lambda_0}, q_{0_0}), a, (q_{\lambda_1}, q_{0_1}))\}$
4. $\hookrightarrow = \{((q_{\lambda_0}, q_{0_0}), a, (q_{\lambda_1}, q_{0_1}))\}$

The semantic of $\mathcal{H}$ is given in Figure 5.2(b).

Remark 5.1. *The semantics of hierarchical transition system is compositional, in other words it is calculated from the leaves to the root of the hierarchy. In consequence, the meaning of a node depends only on its subtree and not on its ancestors in the hierarchy.*

For the sake of clarity, in the forecoming examples of this chapter, we will only represent reachable states (from the initial states) of the semantic of a hierarchical transition system.

Paths

We extend the notion of paths (see Chapter 4) to labeled transition system in the natural way: A *path* in a labeled transition system is any non-empty finite sequence of states and events $q_0, e_1, q_1 \ldots, e_n, q_n$ such that $q_i \xrightarrow{e_i} q_{i+1}$ for $0 \leq i < n$. Such a path is called a *run* if $q_0 \in I$ and $q_n \in F$. The set of all paths (resp. runs) of S is denoted by $Path(S)$ (resp. $Run(S)$). If $\mathcal{H}$ is a hierarchical transition system then we simply write $Path(\mathcal{H})$ instead of $Path([\![\mathcal{H}]\!])$.

Given a hierarchical transition system $\mathcal{H}$ a *projection* of a path $\pi^\flat \in Path(\mathcal{H})$ onto a transition system S_v for some node v of the hierarchy tree t is denoted by $\pi^\flat(v)$ and is defined componentwise: $\pi^\flat(v) = q_0^\flat(v), e_1^\flat(v), q_1^\flat(v) \ldots, q_n^\flat(v)$.

5.1.2 Issues related to priorities

We now discuss the issues that arise when priorities are present in a hierarchical transition system. We will show that we cannot remove priorities. It is not even possible to move them upwards to the root node of the hierarchy.

Consider the hierarchical transition system $\mathcal{H}$ given in Figure 5.3(a), which contains two nodes λ and 0, and their associated transition systems S_λ, and S_0. The inner node 0 has a nontrivial priority relation: $b \preccurlyeq a$. We will show that this priority relation cannot be replaced by a relation in the λ node. Both transition systems use the same alphabet $\Sigma_\lambda = \Sigma_0 = \{a, b, \varepsilon\}$, but the transition system S_λ has no transition labeled with a. The set of synchronizations vectors of the λ node is: $\{(a, a); (b, b); (\varepsilon, \varepsilon)\}$. In other words, the events a, b, and ε of the λ node are synchronized respectively with the events a, b, and ε of the 0 node. The node 0 define the priority: $b \prec a$. In Figure 5.3(b) we have an almost identical hierarchical transition system where we just moved the priority of the node 0 to the node λ. This modification has an immediate effect on semantics (cf. n Figure 5.3(c) and Figure 5.3(d)). This example illustrates that priority resolution when determining the semantic of a hierarchical transition system cannot be delayed in a sense that they cannot be moved to higher nodes.

One could argue, that on this example we can define a new priority relation for the λ node of $\mathcal{H}'$ that would allow us to obtain the desired semantic. For instance the priority

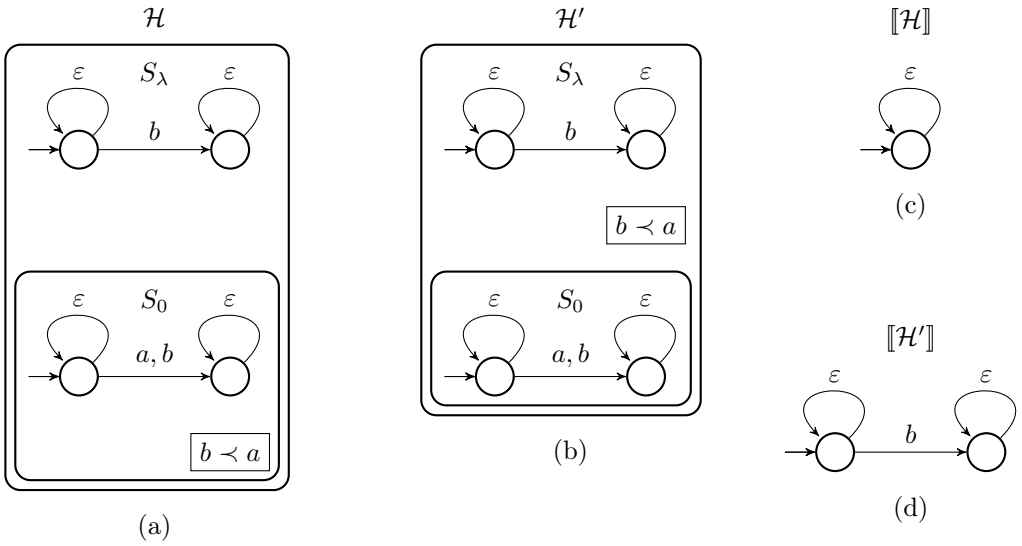


Figure 5.3: An example illustrating the result of delaying priority resolution on a hierarchical transition system, (a) the hierarchical transition system $\mathcal{H}$, (b) the hierarchical transition system $\mathcal{H}'$, (c) $[\![\mathcal{H}]\!]$ the semantic of $\mathcal{H}$, (d) $[\![\mathcal{H}']\!]$ the semantic of $\mathcal{H}'$.

relation could be $b \prec \epsilon$. However, in general we can not always find a suitable priority relation.

Sometimes it is not possible to determine a "global" priority (i.e. a priority relation on the λ node) in order to delay priority resolution. To illustrate this statement consider the hierarchical transition system given in Figure 5.4(a). This hierarchical transition system $\mathcal{H}$, has its counterpart $\mathcal{H}'$ Figure 5.4(b) that we will use to try to determine a global priority (the box containing "$? \preccurlyeq ?$" is the priorities we want to determine) so that we have $[\![\mathcal{H}]\!]$ equal to $[\![\mathcal{H}']\!]$. The λ nodes of the hierarchical transition systems synchronize on common events. In Figure 5.4(c) we have given the semantic of $\mathcal{H}$, and below in Figure 5.4(d) the semantic of $\mathcal{H}'$ without applying any priority.

Consider now $[\![\mathcal{H}']\!]$, let us examine all possible priorities that could be applied in order to have $[\![\mathcal{H}]\!] = [\![\mathcal{H}']\!]$. Note that by construction we only need to determine a priority to apply to $[\![\mathcal{H}']\!]$ to replace the "$? \prec ?$" box. First lets try $b \prec c$ but applying such priority would prune away the bottom of the transition system. Moreover, it is easily seen that any other priority could not prune away the outgoing transition of the initial state labeled with b. We therefore conclude that there does not exists a global priority that gives us $[\![\mathcal{H}]\!] = [\![\mathcal{H}']\!]$ (up to an isomorphism). This shows that we need to allow priorities on each level of a hierarchical systems.

Also note, that we could try to modify the synchronization vectors, in order to eliminate the undesired transition. For instance by deciding that the event b of the λ node of $\mathcal{H}'$ now synchronizes with the c event of the node 0, and we can use $b \prec c$ (or any other priority relation) for $? \prec ?$. This would eliminate the undesired transition, but would

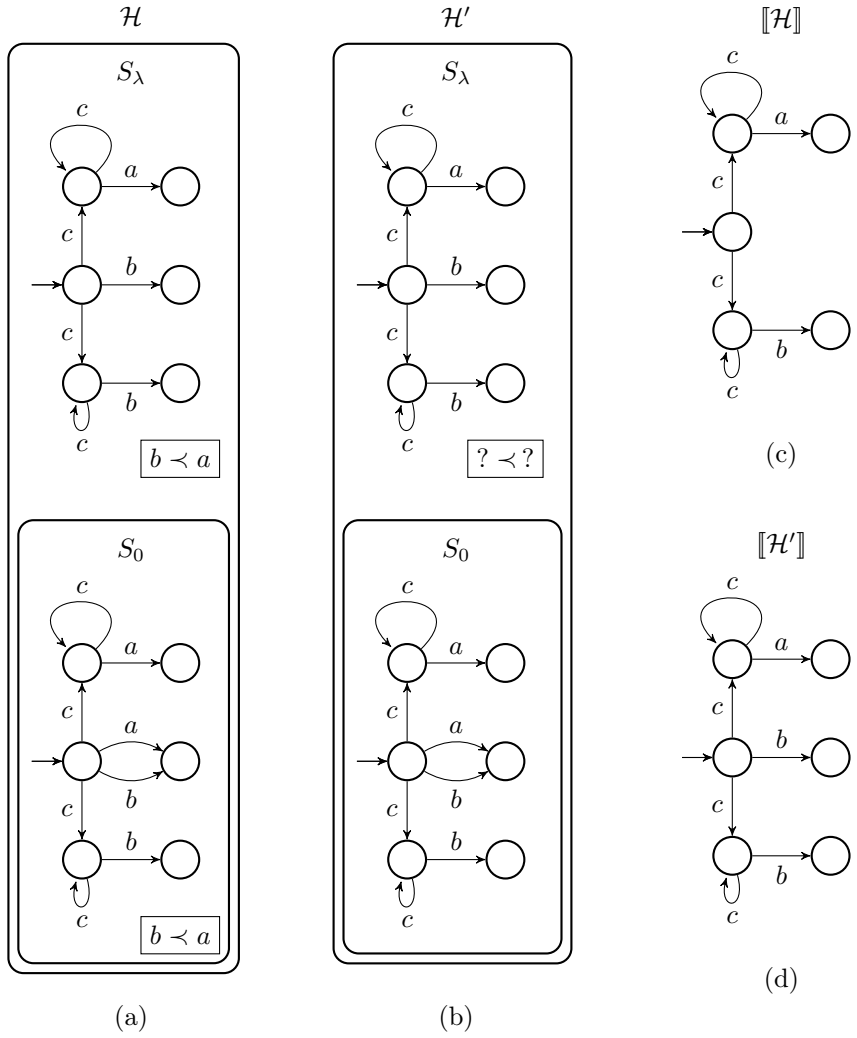


Figure 5.4: An example illustrating the non existence of a priority over the flattened hierarchical transition system, (a) a hierarchical transition system $\mathcal{H}$, (b) $\mathcal{H}'$ the counterpart of $\mathcal{H}$ with a global priority to be determined, (c) the semantic of $\mathcal{H}$, (d) the semantic of $\mathcal{H}'$ before applying any priority relation.

also prune away the bottom part of the semantic of $\mathcal{H}'$.

Even if our setting appears to be quite complicated, it offers a big flexibility and modeling power. In the following we go over some examples presenting usefulness of hierarchical transition system for modular presentation of models.

5.1.3 Advantages of Modular Representation

Modular representations of systems are nowadays a standard in the software industry as well as many other industries. One of the reasons modular representation became a standard is the ever-growing size of modern systems (e.g., railways systems, airplanes,...). Overtime, dividing a system into components, that are responsible for a single functionality became crucial. Such a decomposition allows one to have a better grasp of a the overall system. Yet these much smaller components, once put together still describe a huge system, and need to communicate with each other as well as with their environment. Before presenting our approach to the verification of hierarchical transition systems, we discuss some examples showing advantages of hierarchical transition systems.

Succinctness of Representation

Let us start by a classical example: the counting wheels. The modelization we consider is given as a hierarchical transition system in Figure 5.5. The hierarchical transition system models a three wheels binary counter, where the least significant bit is the rightmost bit. The hierarchical schema $\mathcal{S}$ (Figure 5.5(a)) is the skeleton of our wheel counter and is composed of two types of nodes: wheel nodes: $0, 10, 11$, and interface nodes: $\lambda, 1$.

Wheel nodes simply define an alphabet $\{inc, reset, \varepsilon\}$ where inc stands for the *increment* event, $reset$ is the *reset* event, and ε is our "no operation" event. They do not define any priorities between their events, and dot not synchronize their events with any successor (they do not have successors). The transition system associated to the wheel node is S_b and is given in Figure 5.5(b) (i.e. we have $S_0 = S_{10} = S_{11} = S_b$). The transition system models a bit counter, one state represents the 0 value, and the other represents the 1 value. The inc event changes the state from 0 to 1, $reset$ is the dual transition, and the ε event labels the loops, and allows the system to "nothing" (these transitions are need to allow an asynchronous behavior of the model).

The interface nodes use the same alphabet $\{inc, reset, \varepsilon\}$, do not define a priority relation over their events, but define four synchronization vectors. Before presenting the synchronization vectors, and their impact, let us first introduce the transition system used for these nodes. The nodes λ, and 1 use the transition system S_a given in Figure 5.5(a) (i.e., we have $S_\lambda = S_1 = S_a$). The transition system is simple, it is just a single (initial) state on which all events loop. Now let us go back to the synchronization vectors.

The interface define the following four synchronization vectors:

1. $(\varepsilon, \varepsilon, \varepsilon)$
2. $(\varepsilon, \varepsilon, inc)$

3. $(inc, inc, reset)$

4. $(reset, reset, reset)$

In order to explain these vectors, consider the 1 node (i.e., we are restricting ourselves to $\mathcal{H}_{\downarrow_1}$). This node synchronizes its successors 10 and 11. The first vector $(\varepsilon, \varepsilon, \varepsilon)$ synchronize all ε events. As previously stated, this allows the system to simply do nothing: the ε event of S_1, S_{10}, and S_{11} always loops on their current state. The second vector $(\varepsilon, \varepsilon, inc)$ ensures that when the rightmost counter (least significant bit) increments, the transition systems S_1, and S_{10} must fire an ε labeled transition. The third vector $(inc, inc, reset)$, synchronizes the increment event of the bit of higher order, with the reset event of the lower order bit. Observe, that this synchronization ensures that the low order bit has the value 1, since the unique transition in S_b labeled with $reset$ has for source the state we use to model the value 1. This do not force the low order bit to take the value 1, but forbids the high order bit to increment if the low order bit has not the value 1. Finally, the last vector $(reset, reset, reset)$ forces all transition systems to reset all together. Yet, as previously remarked, each of the synchronized transition systems must be in a state where they can fire a transition labeled $reset$.

It is easily seen that the semantic of $\mathcal{H}_{\downarrow_1}$ describes a two bit counter. Now to extend our counter with an extra bit, it is sufficient to treat $\mathcal{H}_{\downarrow_1}$ as a least significant bit, and interface it with a sibling counter node. This is the three wheels binary counter of Figure 5.5.

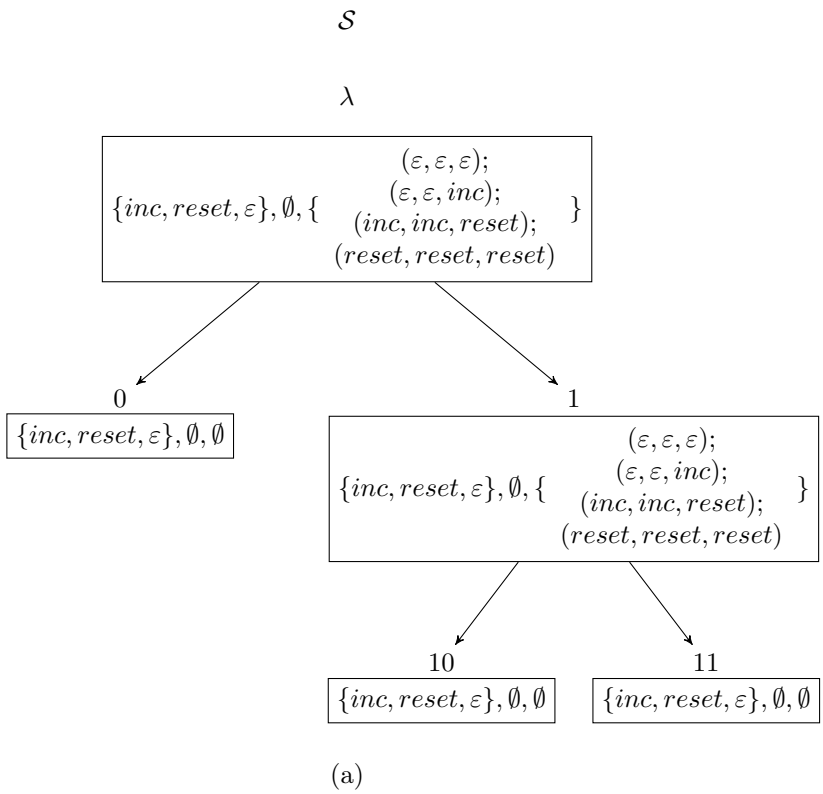
Continuing this way we can construct n-wheel counter in a modular way. To add one wheel we will need to add two nodes: a wheel itself and a link. Of course each wheel doubles the state space of the whole system. This shows that the semantics of a hierarchical system can be exponentially bigger than its description.

Communicating with Synchronization Vectors

Another feature of our hierarchical transition system, is the inter-node communication by synchronization vectors. The use of synchronization vectors allows us to have a succinct representation of a large communication scheme. Note that the underlying setting was originally proposed by Arnold and Nivat: synchronous product of transition systems with synchronization vectors [AN82].

To illustrate this communication mechanism, let us go back to our counting wheel example. In the example our counting wheel behaves without any possible dysfunctions. In Figure 5.6 we have extend our counting wheels with nodes that model the dysfunction of a wheel.

Dysfunction nodes 00, 100, and 110 define an alphabet $\{ok, f_1, f_2, r_1, r_2, \varepsilon\}$ where ok represents the absence of dysfunction, f_1 and f_2 represent two failures that the system can encounter, r_1, r_2 are the corresponding reparations, and ε stands for the "no operation" event. No priority is defined between the events, and do not synchronize their events. The transition system associated to the node is S_c and is given in Figure 5.6(d). The transition system models the impact of failures and reparations: when a failure occurs the



Node	Transition System
λ	S_a
1	
0	S_b
10	
11	

(d)

Figure 5.5: A counting wheels system, (a) The hierarchical schema $\mathcal{S}$, (b) the transition system S_a for a whell link, (c) the transition system S_b modeling a counting wheel, (d) the mapping table matching nodes of t to transition systems.

"ok" labeled transition cannot be fired, and a reparation allows the transition system to reach a state where the "ok" labeled transition can be fired. In our hierarchical transition system we added for each wheel node a dysfunction node as it successor. This will allow us to disable a wheel node whenever a "failure" event occurs. To manage this behavior, wheel nodes have been modified to define two synchronization vectors: (inc, ok) and $(reset, ok)$. These synchronization vectors forbid the wheel node to increment or reset if its associated dysfunction node does not fire an ok labeled transition. Note that these synchronization vectors have a global impact on our counting wheel system. For instance, if the wheels 10 or 11 are dysfunctioning the entire system is blocked: it will only be able to fire the ε transition synchronized with the $(\varepsilon, \varepsilon, \varepsilon)$ vector in the λ node. On the other hand if the 0 wheel is dysfunctioning, it will still be possible to fire the ε transition synchronized with the $(\varepsilon, \varepsilon, \varepsilon)$ or $(\varepsilon, \varepsilon, inc)$ vectors in the λ node.

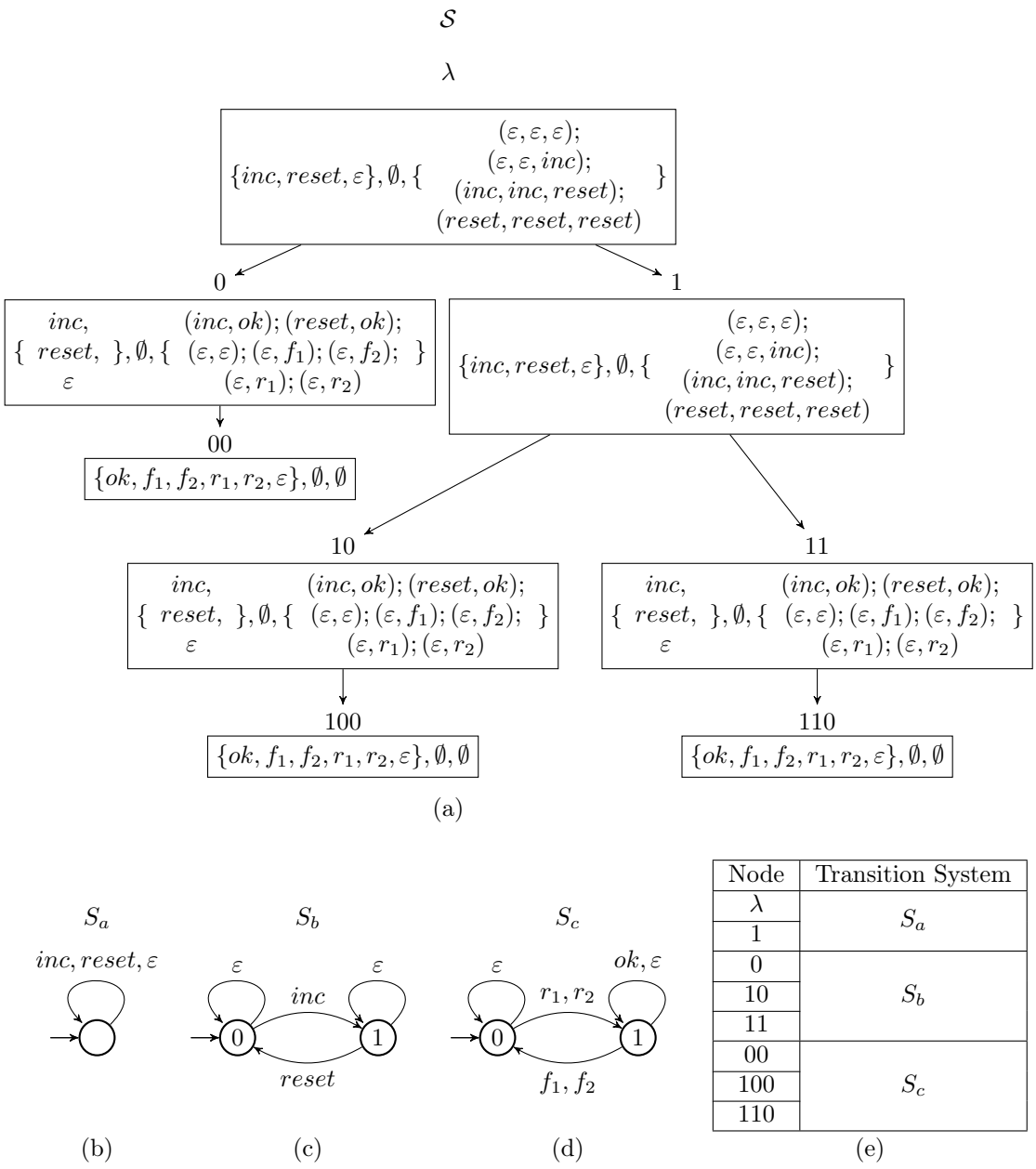
Moreover, the events f_1, f_2, r_1, and r_2 of a dysfunction node are synchronized in a wheel node with the local ε event. This allows us to "mask" the failure events in the predecessor of wheel node. For instance in the 1 node, the synchronization vector $(\varepsilon, \varepsilon, inc)$ will be mapped with each ε synchronization of its 10 successor (e.g. (ε, f_1), (ε, f_2), (ε, r_1), (ε, r_2), and $(\varepsilon, \varepsilon)$). Likewise the $(\varepsilon, \varepsilon, \varepsilon)$ will be mapped with each ε synchronization of its 10 and 11 successors, this will generate 25 new events in $\mathcal{H}_{\downarrow_1}$. Continuing this way, when we had 5 events in the semantic of our dysfunction free counting wheel system (i.e., in $\mathcal{H}_{\downarrow_\lambda}$), we now have 157 events that are induced by a small set of synchronization vectors.

Using Priorities

To illustrate the use of priorities, let us present our queue model[1]. The hierarchical transition system given in Figure 5.7 is our model of a queue. The queue can hold three objects of type a, or b. The hierarchical schema $\mathcal{S}$ (Figure 5.7(a)) is the skeleton of our queue and is composed of two types of nodes: cell nodes: $0, 10, 11$, and interface nodes: $\lambda, 1$.

Cell nodes simply define an alphabet $\{put_a, get_a, put_b, get_b, \varepsilon\}$ where put_a (resp. put_b) represents the action of inserting an object of type a (resp. b) into the cell. The event get_a (resp. get_b) is the dual action: removing an object of type a (resp. b) from the cell. The ε is the "no operation" event. Cell nodes do not define any priorities between their events, and do not synchronize their events with any successor (they do not have successors). The transition system associated to the cell nodes (see Figure 5.7(d)) is S_b and is given in Figure 5.7(b) (i.e. we have $S_0 = S_{10} = S_{11} = S_b$). The transition system models a cell, one state represents the cell containing the object a, another state represents the cell containing the object b, and the initial state represents the cell when it is empty. The events label the transitions in the normal way: to put an object in the cell, it has to be empty. You can only get an object that is present is the cell, and the ε event labels a loop on each state of the transition system.

[1]The model is an hierarchical transition system version of the AltaRica node `FIFO2` given in Figure 2.15(a).



Node	Transition System
λ	S_a
1	
0	S_b
10	
11	
00	S_c
100	
110	

Figure 5.6: A dysfunctionnal counting wheels system, (a) The hierarchical schema $\mathcal{S}$, (b) the transition system S_a for a whell link, (c) the transition system S_b modeling a counting wheel, (d) the transition system S_c modeling a dysfunctional mode, (e) the mapping table matching nodes of t to transition systems.

Interface nodes use the alphabet of the cell nodes together with an extra event: $shift$. We will use this event to move objects from the tail of the queue to its head. Before detailing the priority relation, and synchronization vectors, let us look at the transition system we associated to an interface node. The nodes λ, and 1 use the transition system S_a given in Figure 5.7(a) (i.e., we have $S_\lambda = \Sigma_1 = S_a$). The transition system is simple, it is a single (initial) state on which all events loop.

A queue is a FIFO (First In First Out) container. When an element is taken out of the queue, the remaining elements of the queue move one step higher (i.e., closer to the head of the queue). Therefore, in our model, we need to ensure that the content of a cell is shifted one step higher as soon as possible (i.e., before any other event can occur). Thanks to priorities we can enforce this behavior of our model using a simple statement: $\{put_a, get_a, put_b, get_b\} \prec shift$[2]. Of course, one could ensure this desired behavior without the use of priorities, but this would require a more complex transition system for the interface node: one that would store the current global state of the queue, and enforce shifting object at the right moments.

To finish presentation of our model, we discuss the synchronization vectors. We only present the synchronization vectors related to the objects of type a. The synchronization vectors are:

1. $(put_a, \varepsilon, put_a)$
2. $(get_a, get_a, \varepsilon)$
3. $(shift, put_a, get_a)$

In order to explain these vectors, consider the 1 node (i.e., we are restricting ourselves to $\mathcal{H}_{\downarrow_1}$). This node synchronizes its successors 10 and 11. Recall that, 10 and 11 are both cell nodes. As expected, the first vector $(put_a, \varepsilon, put_a)$ ensures that a new element is inserted into the tail of the queue (here the rightmost cell). Note that if the tail of the queue is not empty, then it is not possible to insert an element in the queue (even if the head is empty). Likewise, popping an object of the queue is done on the head of the queue and this is ensured by the vector $(get_a, get_a, \varepsilon)$. The last vector performs the shift operation of the queue: $(shift, put_a, get_a)$. In other words, if the head cell is empty and the tail cell holds an object, then the shift operation inserts the object into the head cell while popping it from the tail cell. Now, recall that in the interface node 1 we have $\{put_a, get_a\} \prec shift$. This priority forces the model to shift as soon as possible: suppose that we insert an object into the queue, the first synchronization vector ensures that the object is inserted into the tail cell. Since at this point the head cell is empty, the interface node forces the shift event to occur (more precisely, it is the only possible event in that state). This behavior is enforced by the priority $\{put_a, get_a\} \prec shift$. Thanks to this priority, the object is moved to the head of the queue, and can be popped.

Note we can easily extend our queue with a new cell, it suffices to treat $\mathcal{H}_{\downarrow_1}$ as the tail of a queue, and interface it with a new cell node. Doing so we obtain the hierarchical transition system given in Figure 5.7.

[2]Here we use the notation $\{put_a, get_a, put_b, get_b\} \prec shift$ for $put_a \prec shift, \ldots, get_b \prec shift$.

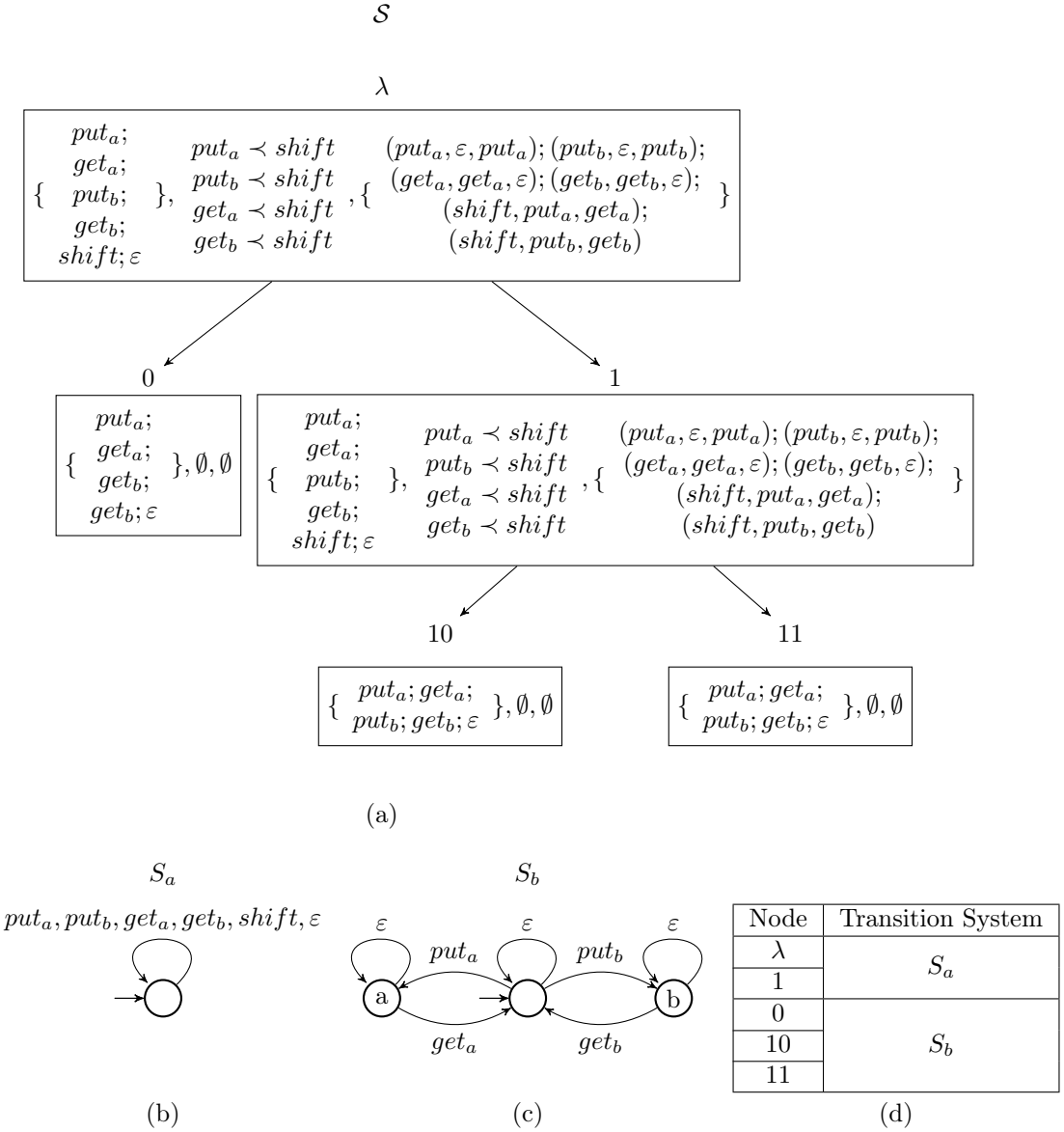


Figure 5.7: An hierarchical transition system modeling a queue, (a) the hierarchical schema $\mathcal{S}$, (b) the transition system S_a linking two queues, (c) the transition system S_b modeling a queue that can contain an object a or an object b, (d) the mapping table that matching nodes of t to transition systems.

5.2 Abstractions for Priority Free Hierarchical Transition Systems

Till now we have used modular representation as a convenient and succinct way of representing transition systems. Usually, a decomposition into modules represents also logical structure of the system. It is tempting then to exploit this structure in CEGAR approach. We make this idea precise by assuming that abstraction should have the same modular structure as the analyzed system. This means that each component of the system is abstracted separately: decomposition reflects the structure of the system.

As we have already seen, priorities are a source of significant complications. These are even more acute in case of hierarchical abstractions. For this reason we first consider priority-free case where the hierarchical approach works smoothly.

The idea is to cover each transition system in the hierarchy separately and hope that the result is still an abstraction. This will be indeed the case if there are not priorities. In order to construct a CEGAR procedure it is also crucial to be able to detect if a path from an abstract system is spurious or not. We show that a hierarchical covering allow to do this without calculating the semantics of the system.

5.2.1 Hierarchical Covering

Since we will work with covers and abstractions we will briefly recall their definitions and properties. The definition of hierarchical covering will be then an easy generalization.

Abstraction, Covers & Refinement

As we have seen in Chapter 3.2.1, the notion of abstraction is usually expressed in terms of simulation [Mil71, LGS+95]. Formally, given two transition systems S_c and S_a, a *simulation relation* from S_c to S_a is any relation $\rho \subseteq (Q_c \times Q_a)$ satisfying, for all $q_c, q_c' \in Q$, $q_a \in Q_a$ and $e \in \Sigma_c$:

$$\text{for all } (q_c, q_a) \in \rho \text{ and } q_c \xrightarrow{e} q_c' \text{ there is } q_a' \in Q_a \text{ such that } (q_a, q_a') \in \rho \text{ and } q_a \xrightarrow{e} q_a'$$

Definition 5.2. *A transition system S_a is an* abstraction *of a transition system S_c if there is a simulation relation ρ from S_c to S_a satisfying the two following properties:*

- *for every $q_c \in I_c$, there is $q_a \in I_a$ with $(q_c, q_a) \in \rho$,*
- *for every $q_c \in F_c$, if $(q_c, q_a) \in \rho$ then $q_a \in F_a$.*

In this case we will also say that S_a is an abstraction of S_c through *ρ.*

Let us first recall some notions that were defined in Chapter 4.1.1. Given a set A, we write $\mathcal{P}^+(A) = \mathcal{P}(A) \setminus \{\emptyset\}$ for the set of non-empty subsets of A. A *cover* of A is any subset C of $\mathcal{P}^+(A)$ such that $A = \bigcup C$. A *covering* of a labeled transition system $S = \langle Q, \Sigma, \rightarrow, I, F\rangle$ is a labeled transition system

$$\widehat{S} = \langle \widehat{Q}, \Sigma, \rightarrow, \widehat{I}, \widehat{F}\rangle$$

such that $\widehat{Q}$ is a cover of Q and, for every $\widehat{q}, \widehat{r} \in \widehat{Q}$ and $e \in \Sigma$:

- $\widehat{q} \xrightarrow{e} \widehat{r}$ iff $q \xrightarrow{e} r$ for some $q \in \widehat{q}$ and $r \in \widehat{r}$,
- $\widehat{q} \in \widehat{I}$ iff $\widehat{q} \cap I \neq \emptyset$,
- $\widehat{q} \in \widehat{F}$ iff $\widehat{q} \cap F \neq \emptyset$.

Hence every state of a covering is a set of states of the original system, and the existence of transition between two states is determined by the elements contained in these states.

Lemma 5.1. *Every covering $\widehat{S}$ of a transition system S is an abstraction of S. The relation witnessing this is the membership relation $\{(q, \widehat{q}) \in Q \times \widehat{Q} \mid q \in \widehat{q}\}$.*

Proof. Let $q \in Q$, $\widehat{q} \in \widehat{Q}$. Assume that $q \in \widehat{q}$ and $q \xrightarrow{e} r$. Since $\widehat{Q}$ is a cover of Q, there exists $\widehat{r} \in \widehat{Q}$ such that $r \in \widehat{r}$. It follows from the definition of coverings that $\widehat{q} \xrightarrow{e} \widehat{r}$. We have thus shown that the membership relation is a simulation relation from S to $\widehat{S}$. Let us prove that it also satisfies the conditions of Definition 5.2. Let $q \in I$. Since $\widehat{Q}$ is a cover of Q, there exists $\widehat{q} \in \widehat{Q}$ such that $q \in \widehat{q}$. As $\widehat{q} \cap I \neq \emptyset$, we get that $\widehat{q} \in \widehat{I}$. Similarly, let $q \in F$ and $\widehat{q} \in \widehat{Q}$ with $q \in \widehat{q}$. As $\widehat{q} \cap F \neq \emptyset$, we get that $\widehat{q} \in \widehat{F}$.

Since the membership relation is a simulation relation from S to $\widehat{S}$ satisfying the conditions of Definition 5.2, we conclude that $\widehat{S}$ is an abstraction of S. □

Remark 5.2. *To emphasize the fact that coverings are abstractions, we will sometimes use the term* cover abstraction*.*

The above lemma shows that, in the case of cover abstractions, the membership relation is a simulation relation. In other words, if $q \in \widehat{q}$ then $\widehat{q}$ simulates q. This justifies the use of the membership relation in the following definition.

Definition 5.3. *Let S be a transition system S, and let $\widehat{S}$ be a cover abstraction of S. A path $\pi = q_0, e_1, q_1, \ldots, e_n, q_n \in Path(S)$ is an* instantiation *of a path $\widehat{\pi} = \widehat{q}_0, e_1, \widehat{q}_1, \ldots, e_n, \widehat{q}_n \in Path(\widehat{S})$ if $q_i \in \widehat{q}_i$ for all $i = 0, \ldots, n$. A path $\widehat{\pi} \in Path(\widehat{S})$ is* feasible *if there is an instantiation of it. Otherwise, $\widehat{\pi}$ is* spurious*.*

Next lemma implies that if there is run, i.e. a path from an initial to a final state, in a concrete system then there is one in its cover.

Lemma 5.2. *If $\widehat{S}$ is a cover abstraction of S, then every path of S is an instantiation of a path of $\widehat{S}$.*

Proof. Consider a path $\pi = q_0, e_1, q_1, \ldots, e_n, q_n$ of S. Since $\widehat{Q}$ is a cover of S, there exists, for every $0 \leq i \leq n$, an abstract state $\widehat{q}_i \in \widehat{Q}$ such that $q_i \in \widehat{q}_i$. It follows from the definition of coverings that $\widehat{q}_{i-1} \xrightarrow{e_i} \widehat{q}_i$ for all $0 < i \leq n$. Therefore, $\widehat{\pi} = \widehat{q}_0, e_1, \widehat{q}_1, \ldots, e_n, \widehat{q}_n$ is a path of $\widehat{S}$. Furthermore, π is obviously an instantiation of $\widehat{\pi}$. □

Yet a cover abstraction may contain spurious paths. A spurious path is a path of an abstraction that is not an instance of any path of the concrete system. If we discover a spurious path in the abstraction, it is necessary to refine it (i.e., compute a less coarse abstraction) in order to eliminate the spurious path.

As a cover abstraction is completely defined by a cover of the set of states, refinement of a cover abstraction is defined in terms of a refinement of covers. The refinement via split pairs, that is introduced in Chapter 4 Section 4.1.2, is a particular instance of the following definition.

Definition 5.4. *A cover $\widehat{Q}_r$ of Q* refines *a cover $\widehat{Q}_a$ of Q, written $\widehat{Q}_r \trianglelefteq \widehat{Q}_a$, if for every $\widehat{q}_r \in \widehat{Q}_r \setminus \widehat{Q}_a$, there exists $\widehat{q}_a \in \widehat{Q}_a \setminus \widehat{Q}_r$ such that $\widehat{q}_r \subset \widehat{q}_a$.*

Definition 5.5. *Consider two cover abstractions $\widehat{S}_a, \widehat{S}_r$ of a transition system S. Let $\widehat{Q}_a$ and $\widehat{Q}_r$ be the sets of states of the two systems. We say that $\widehat{S}_r$* refines *$\widehat{S}_a$, written $\widehat{S}_r \trianglelefteq \widehat{S}_a$, if $\widehat{Q}_r$ refines $\widehat{Q}_a$.*

Very similar argument as that in Lemma 5.1 shows:

Lemma 5.3. *Given two cover abstractions $\widehat{S}_a, \widehat{S}_r$ of S, if $\widehat{S}_r$ refines $\widehat{S}_a$, then $\widehat{S}_a$ is an abstraction of $\widehat{S}_r$. The simulation witness is the containment relation $\subseteq$.*

For finite-state transition systems, refinement enjoys additional properties that are useful for termination analysis of algorithms based on iterative abstraction refinement. The next proposition, may be seen as a generalization of the proof that Cegar and PCegar algorithms from Chapter 4 terminate.

Proposition 5.1. *If S is finite-state, then the refinement relation $\trianglelefteq$ on cover abstractions of S is a well-founded partial order.*

Proof. We show that the refinement relation $\trianglelefteq$ on covers of Q is a well-founded partial order. The relation $\trianglelefteq$ is obviously reflexive. To prove antisymmetry, assume that $\widehat{Q}_2 \trianglelefteq \widehat{Q}_1 \trianglelefteq \widehat{Q}_2$. For every $i, j \in \{1, 2\}$, it holds, by definition, that:

$$\forall \widehat{q}_i \in \widehat{Q}_i \setminus \widehat{Q}_j \cdot \exists \widehat{q}_j \in \widehat{Q}_j \setminus \widehat{Q}_i \cdot \widehat{q}_i \subset \widehat{q}_j$$

Since Q is finite, $(\mathcal{P}(Q), \subseteq)$ satisfies the ascending chain condition. We derive that $\widehat{Q}_1 \setminus \widehat{Q}_2 = \widehat{Q}_2 \setminus \widehat{Q}_1 = \emptyset$, which entails that $\widehat{Q}_1 = \widehat{Q}_2$.

The proof that $\trianglelefteq$ is transitive is similar to the antisymmetry proof. Assume that $\widehat{Q}_3 \trianglelefteq \widehat{Q}_2 \trianglelefteq \widehat{Q}_1$. It holds, by definition, that:

$$\begin{cases} \forall \widehat{q} \in \widehat{Q}_2 \setminus \widehat{Q}_1 \cdot \exists \widehat{r} \in \widehat{Q}_1 \setminus \widehat{Q}_2 \cdot \widehat{q} \subset \widehat{r} \\ \forall \widehat{q} \in \widehat{Q}_3 \setminus \widehat{Q}_2 \cdot \exists \widehat{r} \in \widehat{Q}_2 \setminus \widehat{Q}_3 \cdot \widehat{q} \subset \widehat{r} \end{cases}$$

Observe that $\widehat{Q}_i \setminus \widehat{Q}_j \subseteq (\widehat{Q}_i \setminus \widehat{Q}_k) \cup (\widehat{Q}_k \setminus \widehat{Q}_j)$ for every $i, j, k \in \{1, 2, 3\}$. It follows that:

$$\begin{cases} \forall \widehat{q} \in \widehat{Q}_2 \setminus \widehat{Q}_1 \cdot \exists \widehat{r} \in \Big((\widehat{Q}_1 \setminus \widehat{Q}_3) \cup (\widehat{Q}_3 \setminus \widehat{Q}_2)\Big) \cdot \widehat{q} \subset \widehat{r} \\ \forall \widehat{q} \in \widehat{Q}_3 \setminus \widehat{Q}_2 \cdot \exists \widehat{r} \in \Big((\widehat{Q}_1 \setminus \widehat{Q}_3) \cup (\widehat{Q}_2 \setminus \widehat{Q}_1)\Big) \cdot \widehat{q} \subset \widehat{r} \end{cases}$$

Since $(\mathcal{P}(Q), \subseteq)$ satisfies the ascending chain condition, the following assertions follows:

$$\forall \widehat{q} \in \Big((\widehat{Q}_2 \setminus \widehat{Q}_1) \cup (\widehat{Q}_3 \setminus \widehat{Q}_2\Big) \cdot \exists \widehat{r} \in (\widehat{Q}_1 \setminus \widehat{Q}_3) \cdot \widehat{q} \subset \widehat{r}$$

The observation that $\widehat{Q}_1 \setminus \widehat{Q}_3 \subseteq (\widehat{Q}_2 \setminus \widehat{Q}_1) \cup (\widehat{Q}_3 \setminus \widehat{Q}_2)$ entails that $\widehat{Q}_3 \trianglelefteq \widehat{Q}_1$. This concludes the proof that $\trianglelefteq$ is a partial order on the set of all covers of Q. Furthermore, since this set is finite, we obtain that $\trianglelefteq$ is well-founded. □

Covers for Hierarchical Transition Systems

Since covers are practical abstractions for transition systems, we are going to generalize their definition to hierarchical systems. We will simply cover a hierarchical system componentwise. It turns out that, when there are no priorities, this straightforward approach preserves all good properties of covers. For the remainder of this section, we consider a hierarchical transition system $\mathcal{H} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}, \{S_v\}_{v\in t}\rangle$.

Definition 5.6. *A* hierarchical covering *of* $\mathcal{H} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}, \{S_v\}_{v\in t}\rangle$ *is a hierarchical transition system* $\widehat{\mathcal{H}} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}, \{\widehat{S}_v\}_{v\in t}\rangle$*, such that* $\widehat{S}_v$ *covers* S_v *for each* $v \in t$*.*

We also say that $\widehat{\mathcal{H}}$ *hierarchically covers* $\mathcal{H}$ when $\widehat{\mathcal{H}}$ is a hierarchical covering of $\mathcal{H}$. The semantics of $\mathcal{H}$ and $\widehat{\mathcal{H}}$ are given, according to Definition 5.1, by the transition systems $S_v^\flat = \langle Q_v^\flat, \Sigma_v^\flat, \hookrightarrow_v, I_v^\flat, F_v^\flat\rangle$ and $\widehat{S}_v^\flat = \langle \widehat{Q}_v^\flat, \Sigma_v^\flat, \hookrightarrow_v, \widehat{I}_v^\flat, \widehat{F}_v^\flat\rangle$. Remark that an abstract state $\widehat{q} \in \widehat{Q}_v^\flat$ is *not* a subset of $Q_v^\flat$. However, it will be convenient to view it as such. So we shall identify $\widehat{q}$ with the set $\{q \in Q_v^\flat \mid \forall u \in t{\downarrow_v} \cdot q(u) \in \widehat{q}(u)\}$. With this view, $\widehat{Q}_v^\flat$ is a cover of $Q_v^\flat$.

Analogously to the abstraction of hierarchical systems, we refine hierarchical coverings componentwise. A *refinement* of a hierarchical covering $\widehat{\mathcal{H}_a} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v \}_{v\in t}, \{\delta_v\}_{v\in t}, \{\widehat{S_{a_v}}\}_{v\in t}\rangle$ is the hierarchical covering $\widehat{\mathcal{H}_r} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}, \{\widehat{S_{r_v}}\}_{v\in t}\rangle$ where $\widehat{S_{r_v}}$ refines $\widehat{S_{a_v}}$ for each node v of the three t.

One would expect that hierarchical covering gives a cover abstraction. This may not be the case in the presence of priorities (cf. example in Section 5.3). Fortunately, when a hierarchical system is priority-free everything works out nicely.

Lemma 5.4. *If* $\mathcal{H}$ *is priority-free then for every* $v \in t$*,* $q, r \in Q_v^\flat$ *and* $e \in \Sigma_v^\flat$*:*

$$q \overset{e}{\hookrightarrow}_v r \quad \textit{iff} \quad \forall u \in t{\downarrow_v} \cdot (q(u) \overset{e(u)}{\to}_{vu} r(u))$$

Proof. Since $\mathcal{H}$ is priority-free, the transition relation $\overset{e}{\hookrightarrow}_v$ of its semantics $[\![\mathcal{H}]\!]$, given in Definition 5.1, may be reformulated as follows:

$$q \overset{e}{\hookrightarrow}_v r \quad \text{iff} \quad q(\lambda) \overset{e(\lambda)}{\to}_v r(\lambda) \text{ and } \forall i \in \mathbb{N} \cdot (vi \in t \Rightarrow q{\downarrow_i} \overset{e{\downarrow_i}}{\hookrightarrow}_{vi} r{\downarrow_i})$$

The lemma follows by structural induction on the tree t. □

Lemma 5.5. *If $\widehat{\mathcal{H}}$ is a hierarchical covering of $\mathcal{H}$, and $\mathcal{H}$ is priority-free, then $[\![\widehat{\mathcal{H}}]\!]$ is a cover abstraction of $[\![\mathcal{H}]\!]$.*

Proof. Recall that $[\![\widehat{\mathcal{H}}]\!]$ is $\widehat{S}^\flat_\lambda$, and similarly $[\![\mathcal{H}]\!]$ is $S^\flat_\lambda$. Since $\widehat{\mathcal{H}}$ is a hierarchical covering of $\mathcal{H}$, $\widehat{S}_v$ is a cover abstraction of S_v, for every $v \in t$. Recall that $\widehat{Q}^\flat_\lambda$ is a cover of $Q^\flat_\lambda$. Let us first prove that the initial abstract states of $[\![\widehat{\mathcal{H}}]\!]$ are those of the cover abstraction induced by $\widehat{Q}^\flat_\lambda$. For every $\widehat{q} \in \widehat{Q}^\flat_\lambda$, it holds that:

$$\begin{aligned}
\widehat{q} \in \widehat{I}^\flat_\lambda &\Leftrightarrow \forall v \in t \cdot (\widehat{q}(v) \in \widehat{I}_v) \\
&\Leftrightarrow \forall v \in t \cdot \exists q_v \in \widehat{q}(v) \cdot (q_v \in I_v) \\
&\Leftrightarrow \exists q \in \widehat{q} \cdot \forall v \in t \cdot (q(v) \in I_v) \\
&\Leftrightarrow \exists q \in \widehat{q} \cdot q \in I^\flat_\lambda
\end{aligned}$$

We obtain that $\widehat{q} \in \widehat{I}^\flat_\lambda$ if and only if $\widehat{q} \cap I^\flat_\lambda \neq \emptyset$. This shows that initial abstract states of $[\![\widehat{\mathcal{H}}]\!]$ are those of the cover abstraction induced by $\widehat{Q}^\flat_\lambda$. A similar proof, but with an existential quantification $\exists v \in t$ instead of the universal quantification $\forall v \in t$, shows that final abstract states of $[\![\widehat{\mathcal{H}}]\!]$ are those of the cover abstraction induced by $\widehat{Q}^\flat_\lambda$.

To conclude the proof of the lemma, we must show that the transition relation $\overset{e}{\hookrightarrow}_\lambda$ of $[\![\widehat{\mathcal{H}}]\!]$ satisfies $\widehat{q} \overset{e}{\hookrightarrow}_\lambda \widehat{r}$ if and only if $q \overset{e}{\hookrightarrow}_\lambda r$ for some $q \in \widehat{q}$ and $r \in \widehat{r}$. When $\mathcal{H}$ is priority-free, this is easily shown with Lemma 5.4, as follows. For every $\widehat{q}, \widehat{r} \in \widehat{Q}^\flat_\lambda$ and $e \in \Sigma^\flat_v$,

$$\begin{aligned}
\widehat{q} \overset{e}{\hookrightarrow}_\lambda \widehat{r} &\Leftrightarrow \forall v \in t \cdot (\widehat{q}(v) \overset{e(v)}{\to}_v \widehat{r}(v)) \\
&\Leftrightarrow \forall v \in t \cdot \exists (q_v, r_v) \in (\widehat{q}(v) \times \widehat{r}(v)) \cdot (q_v \overset{e(v)}{\to}_v r_v) \\
&\Leftrightarrow \exists (q, r) \in (\widehat{q} \times \widehat{r}) \cdot \forall v \in t \cdot (q(v) \overset{e(v)}{\to}_v r(v)) \\
&\Leftrightarrow \exists (q, r) \in (\widehat{q} \times \widehat{r}) \cdot q \overset{e}{\hookrightarrow}_\lambda r
\end{aligned}$$

This concludes the proof that $[\![\widehat{\mathcal{H}}]\!]$ is a cover abstraction of $[\![\mathcal{H}]\!]$. □

This lemma implies that every path of $[\![\mathcal{H}]\!]$ is an instantiation of a path of $[\![\widehat{\mathcal{H}}]\!]$. Hence, if $[\![\mathcal{H}]\!]$ has a run then so does $[\![\widehat{\mathcal{H}}]\!]$. Naturally, we will need also to understand the converse situation: when an abstract path has an instantiation.

5.2.2 Abstract path feasibility

As we have noted above, not every path in a hierarchical covering is necessarily feasible. It may be spurious: may not correspond to any path in the concrete system. To check if an abstract path is feasible we can of course calculate the semantics of a hierarchical transition system and play the path in this semantics. Calculating the semantics is however an expensive operation. The advantage of doing CEGAR on hierarchical systems is precisely to avoid computing the semantics explicitly. The following lemma will allow us to check if a path is spurious just by looking separately at each component of the hierarchical transition system.

VerifyHierarchicalPath $(\mathcal{H}, \widehat{\pi})$
Input: $\mathcal{H}$ a Hierarchical Transition System,
$\widehat{\pi}$ a path of an abstraction of $\mathcal{H}$.
1 $X \leftarrow \emptyset$
2 `for` each node v of $\mathcal{H}$ `do`
3 `if` $\widehat{\pi}(v)$ is spurious `for` S_v `then`
4 $X \leftarrow X \cup \{v\}$
5 `done`
6 `return` X

Figure 5.8: The algorithm VerifyHierarchicalPath.

Lemma 5.6. *If $\widehat{\mathcal{H}}$ is a hierarchical covering of $\mathcal{H}$, and $\mathcal{H}$ is priority-free, then for every abstract path $\widehat{\pi}$ in $[\![\widehat{\mathcal{H}}]\!]$, it holds that $\widehat{\pi}$ is feasible in $[\![\mathcal{H}]\!]$ if and only if $\widehat{\pi}(v)$ is feasible in S_v for every node v of $\mathcal{H}$.*

Proof. Consider an abstract path $\widehat{\pi} = \widehat{q}_0, e_1, \widehat{q}_1, \ldots, e_n, \widehat{q}_n$ in $[\![\widehat{\mathcal{H}}]\!]$ that is feasible in $[\![\mathcal{H}]\!]$. There exists a path $q_0, e_1, q_1, \ldots, e_n, q_n$ in $[\![\mathcal{H}]\!]$ such that $q_i \in \widehat{q}_i$ for all $0 \leq i \leq n$. Since $\mathcal{H}$ is priority-free, Lemma 5.4 entails that, for every $0 < i \leq n$ and for every node v of $\mathcal{H}$, $(q_{i-1}(v) \overset{e_i(v)}{\rightarrow}_v q_i(v))$. This means that $\widehat{\pi}(v)$ is feasible in S_v, for every node v of $\mathcal{H}$.

Conversely, assume that, for every node v of $\mathcal{H}$, $\widehat{\pi}(v)$ is feasible in S_v. For each node v of $\mathcal{H}$, there exists a path $q_{v,0}, e_1(v), q_{v,1}, \ldots, e_n(v), q_{v,n}$ in S_v such that $q_{v,i} \in \widehat{q}_i(v)$ for all $0 \leq i \leq n$. Let us define $q_i \in Q^{\flat}_v$, for $0 \leq i \leq n$, by $q_i(v) = q_{v,i}$. Observe that $q_i \in \widehat{q}_i$ for all $0 \leq i \leq n$. Lemma 5.4 entails that $q_{i-1} \overset{e_i}{\hookrightarrow}_\lambda q_i$ for every $0 < i \leq n$. It follows that $\widehat{\pi}$ is feasible in $[\![\mathcal{H}]\!]$. □

Corollary 5.1. *If $\widehat{\mathcal{H}}$ is a hierarchical covering of $\mathcal{H}$, and $\mathcal{H}$ is priority-free, then for every abstract path $\widehat{\pi}$ in $[\![\widehat{\mathcal{H}}]\!]$, it holds that $\widehat{\pi}$ is spurious in $[\![\mathcal{H}]\!]$ if and only if for some node v of $\mathcal{H}$, the path $\widehat{\pi}(v)$ is spurious in S_v.*

This corollary can be translated directly into algorithm VerifyHierarchicalPath presented in Figure 5.8. This algorithm determines the feasibility of an abstract path $\widehat{\pi}$. To do so, the algorithm tests the feasibility of the projection of $\widehat{\pi}$ on each node of $\mathcal{H}$. This test in Line 3 can be done using the methods described in Chapter 3.2.2, like for instance the algorithm VerifyPath of Figure 3.1. The set X keeps track of each node v on which the path $\widehat{\pi}(v)$ is spurious. The algorithm returns the set of nodes for which their projection on $\widehat{\pi}$ is spurious. If the algorithm returns $\emptyset$ then the abstract path $\widehat{\pi}$ is feasible on $\mathcal{H}$, otherwise it is spurious.

These observations are summarized in the following proposition.

Proposition 5.2. VerifyHierarchicalPath *algorithm is correct and terminates.*

Observe that it is enough to return a single node that satisfy the test Line 3 and still have a correct algorithm.We will use it as a part of our CEGAR loop that we are going to present next.

HierarchicalCegar $\left(H, \widehat{H}\right)$
Input: $\mathcal{H}$ a Hierarchical Transition System,
$\widehat{\mathcal{H}}$ a Hierarchical Transition System that covers $\mathcal{H}$.

```
1  while Run([[Ĥ]]) ≠ ∅ do
2     Pick π̂ in Run([[Ĥ]])
3     if π̂ is feasible then
4        return Run([[H]]) ≠ ∅
5     else
6        Pick Ŝ_v such that π̂(v) is spurious
7        Refine Ŝ_v
8  done
9  return "Run([[H]]) = ∅"
```

Figure 5.9: Hierarchical CEGAR algorithm

5.2.3 Hierarchical CEGAR

We now present our CEGAR framework for hierarchical transition systems. Given a hierarchical system $\mathcal{H}$ we want to check if it has a run, i.e., if $Run([\![\mathcal{H}]\!])$ is not empty. We will assume that we are given an initial hierarchical covering $\widehat{\mathcal{H}}$ of $\mathcal{H}$. The algorithm first checks if there is a run in $[\![\widehat{\mathcal{H}}]\!]$. If not then by Lemma 5.5 there is no run in $[\![\mathcal{H}]\!]$ too. Otherwise an abstract run $\widehat{\pi}$ is picked from $Run([\![\widehat{\mathcal{H}}]\!])$. Then $\widehat{\pi}$ is analyzed to determine if it is feasible or spurious. If it is feasible the procedure terminates and returns "$Run([\![\widehat{\mathcal{H}}]\!]) \neq \emptyset$". Otherwise, if $\widehat{\pi}$ is spurious then thanks to Corollary 5.1 it is spurious in one of the components. In this case one of the transition systems $\widehat{S}_u$ of $\widehat{H}$ for which $\widehat{\pi}(u)$ is spurious is chosen. This abstraction is then replaced by one of its refinement who does not contain $\widehat{\pi}(u)$ as one of its runs. For this we can use any of the standard methods c.f. Chapter 3.2.3. Having eliminated a potential counter-example we repeat the loop. The algorithm is presented in Figure 5.9.

Correctness of the HierarchicalCegar procedure follows from the fact that the semantics of a hierarchical covering is indeed an abstraction of the semantics of the concrete system (Lemma 5.5). This means that if there is no run in the semantics of the hierarchical covering then there is none in the semantics of the concrete hierarchical system either.

Termination of the HierarchicalCegar procedure is straightforward for finite hierarchical transition systems thanks to Proposition 5.1. Since each component of the hierarchy is finite, a sufficiently long sequence of refinements will lead to an abstraction that is isomorphic to the initial transition system. When this hierarchical abstraction is reached the procedure will terminate in a single pass through the main loop.

These observations are summarized in the following proposition.

Proposition 5.3. HierarchicalCegar *algorithm is correct and terminates.*

Summarizing, if a hierarchical transition system does not have priorities then cover abstractions allow a smooth implementation of the CEGAR method. We not only avoid

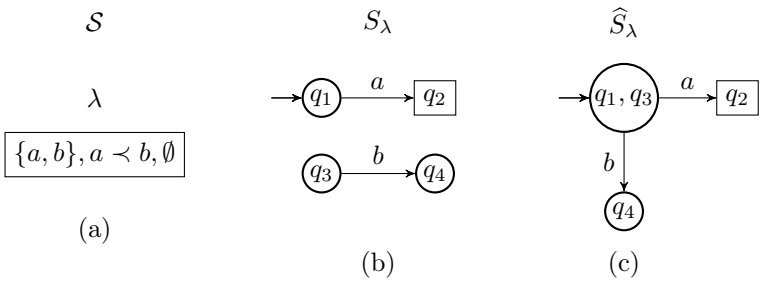


Figure 5.10: Unsoundness of cover abstraction in the presence of priorities, (a) The hierarchical schema $\mathcal{S}$, (b) the transition systems S_λ, (c) $\widehat{S}_\lambda$ a cover abstraction of S_λ.

computing the semantics of the hierarchical system but we are also able to do refinements and feasibility checks locally.

5.3 Hierarchical Transition Systems with priorities

In this section we focus on the impact of priorities in hierarchical transition systems. Our objective is to give a CEGAR algorithm that does not need to calculate complete semantics of a hierarchical system. Recall from Section 5.1.2 that priorities cannot be simply eliminated or moved to the root node of the hierarchy. Of course every transition system can be in fine presented as such, so neither hierarchy nor priorities are indispensable. Yet, as the examples in Section 5.1.3 show, in some cases priorities allow for succinct and elegant presentations. This said the semantics of hierarchical system with priorities is quite involved and one can expect that hierarchical coverings may not work in this case.

After giving an example of problems caused by priorities we will revisit the semantics of hierarchical transition systems. We give a characterization when a hierarchical system is an abstraction of another. Unfortunately the conditions of the characterization will be not easy verify. This is not surprising given the complexity priorities may induce. In the next subsection we follow another route and give a simple sufficient condition in the form of the concept of neat cover. This condition on hierarchical coverings allows us to recover most of the good properties from the priority-free case. After stating necessary properties we will present a CEGAR approach using neat covers. It will turn out that the same algorithm as in hierarchy-free case works, provided we start from a neat cover abstraction.

To motivate this section we give a simple example showing why hierarchical coverings cannot be directly used in the presence of priorities.

Recall that, for "flat" transition systems, coverings are abstractions (cf, Section 5.2.1). However, this is not the case for hierarchical coverings, due to priorities. Consider for instance the hierarchical transition system $\mathcal{H}$ consisting of a single node λ, with alphabet $\{a, b\}$, priority $a \prec b$ and local transition system $S_\lambda = \langle \{q_1, q_2, q_3, q_4\}, \{a, b\}, \{q_1 \xrightarrow{a} q_2, q_3 \xrightarrow{b} q_4\}, \{q_1\}, \{q_2\} \rangle$ given in Figure 5.10(a) and Figure 5.10(b). The semantics of $\mathcal{H}$ is

isomorphic to S_λ, and, in particular, it contains the run q_1, a, q_2. Now, consider the partition $\{\widehat{q}_{13}, \widehat{q}_2, \widehat{q}_4\}$ where $\widehat{q}_{13} = \{q_1, q_3\}$, $\widehat{q}_2 = \{q_2\}$ and $\widehat{q}_4 = \{q_4\}$. The hierarchical covering $\widehat{\mathcal{H}}$ is obtained by replacing S_λ by its cover abstraction $\widehat{S}_\lambda = \langle \{\widehat{q}_{13}, \widehat{q}_2, \widehat{q}_4\}, \{a, b\}, \{\widehat{q}_{13} \xrightarrow{a} \widehat{q}_2, \widehat{q}_{13} \xrightarrow{b} \widehat{q}_4\}, \{\widehat{q}_{13}\}, \{\widehat{q}_2\}\rangle$ given in Figure 5.10(c). Due to the priority $a \prec b$, the transition $\widehat{q}_{13} \xrightarrow{a} \widehat{q}_2$ disappears in the semantics, and, therefore, $[\![\widehat{\mathcal{H}}]\!]$ contains no run. We have thus shown the following proposition.

Proposition 5.4. *There exists a hierarchical transition system $\mathcal{H}$ and a hierarchical covering $\widehat{\mathcal{H}}$ of $\mathcal{H}$ such that $[\![\widehat{\mathcal{H}}]\!]$ is not an abstraction of $[\![\mathcal{H}]\!]$.*

In this example we have not even used the hierarchy, but one can imagine that this example is a part of a hierarchical system. As we have seen in Section 5.1.2, there does not seem to be an easy way to eliminate priorities. In consequence there seem to be no easy way to avoid the problem presented here.

5.3.1 A sufficient condition for being an abstraction

Let us revisit the semantics of hierarchical transition systems with priorities. We will see that its complexity can be captured in the problem of determining the set of outgoing actions from every state. From this we will deduce a sufficient condition for a hierarchical system to be an abstraction of another.

Given a transition system $S = \langle Q, \Sigma, \to, I, F\rangle$ and a state $q \in Q$, the set of *outgoing actions* from q in S is

$$out_S(q) = \{e \in \Sigma \mid \exists r \in Q.\ q \xrightarrow{e} r\}.$$

We shall simply write $out(q)$ when the transition system S is understood from the context.

For the remainder of this section, we consider a hierarchical transition system $\mathcal{H} = \langle t, \{\Sigma_v\}_{v\in t}, \{\preccurlyeq_v\}_{v\in t}, \{\delta_v\}_{v\in t}, \{S_v\}_{v\in t}\rangle$. According to Definition 5.1 its semantics is given by the family of transition systems $S_v^\flat = \langle Q_v^\flat, \Sigma_v^\flat, \hookrightarrow_v, I_v^\flat, F_v^\flat\rangle$, for $v \in t$. We introduce the preorder $\preccurlyeq_v^\flat$ on $\Sigma_v^\flat$ defined by:

$$e \preccurlyeq_v^\flat f \qquad \text{iff} \qquad e(\lambda) \preccurlyeq_v f(\lambda)$$

Observe that $\preccurlyeq_v^\flat$ compares only the parts of the labels coming from Σ_v. In particular it coincides with $\preccurlyeq_v$ when v is a leaf. If X is a set of actions from $\Sigma_v^\flat$, we will write $\mathrm{Max}_{\preccurlyeq_v^\flat}(X)$ for the set of maximal elements in this preorder:

$$\mathrm{Max}_{\preccurlyeq_v^\flat}(X) = \{e \in X : \forall e' \in X.\ e \preccurlyeq_v^\flat e' \Rightarrow e' \preccurlyeq_v^\flat e\}$$

Recall (cf. Definition 5.1) that the semantics of a hierarchical systems is defined using $\overset{e}{\hookrightarrow}_v$ relation. It in turn refers to $\to_v$ relation that is the transition relation of the

system in node v, and to auxiliary relation $\overset{e}{\rightsquigarrow}_v$. Using the notion of maximal elements the definition of the transition relation $\overset{e}{\hookrightarrow}_v$ may be reformulated as follows:

$$q \overset{e}{\hookrightarrow}_v r \quad \text{iff} \quad q \overset{e}{\rightsquigarrow}_v r \text{ and } e \in \mathrm{Max}_{\preccurlyeq_v^\flat} \left\{ d \in \Sigma_v^\flat \;\middle|\; \exists p \in Q_v^\flat \cdot q \overset{d}{\rightsquigarrow}_v p \right\} \tag{5.1}$$

$$q \overset{e}{\rightsquigarrow}_v r \quad \text{iff} \quad q(\lambda) \overset{e(\lambda)}{\rightarrow}_v r(\lambda) \text{ and } \forall i \in \mathbb{N} \cdot (vi \in t \Rightarrow q{\downarrow_i} \overset{e{\downarrow_i}}{\hookrightarrow}_{vi} r{\downarrow_i}) \tag{5.2}$$

Maximal elements can be also used to give a direct characterization of the set $out_{S_v^\flat}(q) = \{e \in \Sigma_v^\flat \mid \exists p \in Q_v^\flat \cdot q \overset{e}{\hookrightarrow}_v p\}$ of outgoing actions from $q \in Q_v^\flat$.

Lemma 5.7. *For every $v \in t$ and $q \in Q_v^\flat$:*

$$\begin{aligned} out_{S_v^\flat}(q) &= \mathrm{Max}_{\preccurlyeq_v^\flat} \left\{ d \in \Sigma_v^\flat \;\middle|\; \exists p \in Q_v^\flat \cdot q \overset{d}{\rightsquigarrow}_v p \right\} \\ &= \mathrm{Max}_{\preccurlyeq_v^\flat} \left\{ d \in \Sigma_v^\flat \;\middle|\; d(\lambda) \in out_{S_v}(q(\lambda)) \wedge \forall i \in \mathbb{N} \cdot (vi \in t \Rightarrow d{\downarrow_i} \in out_{S_{vi}^\flat}(q{\downarrow_i})) \right\} \end{aligned}$$

Proof. It follows from (5.1) that, for every $e \in \Sigma_v^\flat$,

$$\begin{aligned} e \in out_{S_v^\flat}(q) &\Leftrightarrow \exists p \in Q_v^\flat \cdot q \overset{e}{\hookrightarrow}_v p \\ &\Leftrightarrow (\exists p \in Q_v^\flat \cdot q \overset{e}{\rightsquigarrow}_v p) \wedge e \in \mathrm{Max}_{\preccurlyeq_v^\flat} \left\{ d \in \Sigma_v^\flat \;\middle|\; \exists p \in Q_v^\flat \cdot q \overset{d}{\rightsquigarrow}_v p \right\} \\ &\Leftrightarrow e \in \mathrm{Max}_{\preccurlyeq_v^\flat} \left\{ d \in \Sigma_v^\flat \;\middle|\; \exists p \in Q_v^\flat \cdot q \overset{d}{\rightsquigarrow}_v p \right\} \end{aligned}$$

Furthermore, we derive from (5.2) that, for every $d \in \Sigma_v^\flat$ and $p \in Q_v^\flat$,

$$\begin{aligned} \exists p \in Q_v^\flat \cdot q \overset{d}{\rightsquigarrow}_v p &\Leftrightarrow \exists p \in Q_v^\flat \cdot \left(q(\lambda) \overset{d(\lambda)}{\rightarrow}_v p(\lambda) \wedge \forall i \in \mathbb{N} \cdot (vi \in t \Rightarrow q{\downarrow_i} \overset{e{\downarrow_i}}{\hookrightarrow}_{vi} p{\downarrow_i}) \right) \\ &\Leftrightarrow d(\lambda) \in out_{S_v}(q(\lambda)) \wedge \forall i \in \mathbb{N} \cdot (vi \in t \Rightarrow d{\downarrow_i} \in out_{S_{vi}^\flat}(q{\downarrow_i})) \end{aligned}$$

This concludes the proof of the lemma. □

Corollary 5.2. *For every $v \in t$ and $q_1, q_2 \in Q_v^\flat$, if $out_{S_{vu}}(q_1(u)) = out_{S_{vu}}(q_2(u))$ for all $u \in t{\downarrow_v}$, then $out_{S_v^\flat}(q_1) = out_{S_v^\flat}(q_2)$.*

Proof. By structural induction on the tree t. Both the basis and the induction step follow from Lemma 5.7. □

Remark 5.3. *It follows from Lemma 5.7 that equivalence (5.1) may be written as:*

$$q \overset{e}{\hookrightarrow}_v r \quad \textit{iff} \quad e \in out_{S_v^\flat}(q) \textit{ and } q \overset{e}{\rightsquigarrow}_v r. \tag{5.3}$$

The main difficulty in the semantics of hierarchical transition systems comes from priorities. Indeed, without priorities (i.e., when each partial order $\preccurlyeq_v$ is the equality over Σ_v), we may determine whether $q \overset{e}{\hookrightarrow}_v q'$ by looking at each node of the hierarchy $t{\downarrow_v}$ independently. The following lemma shows that we recover this property when the hierarchical event e is known to be an outgoing action. In other words, the complexity in the semantics is captured by the sets $out_{S_v^\flat}(q)$.

Lemma 5.8. *For every $v \in t$, $q, r \in Q^\flat_v$ and $e \in \Sigma^\flat_v$:*

$$q \overset{e}{\hookrightarrow}_v r \quad \textit{iff} \quad e \in out_{S^\flat_v}(q) \textit{ and } \forall u \in t{\downarrow_v} \cdot (q(u) \overset{e(u)}{\to}_{vu} r(u))$$

Proof. By structural induction on the tree t. First, observe that (5.2) and (5.3) entail the following equivalence:

$$q \overset{e}{\hookrightarrow}_v r \quad \Leftrightarrow \quad e \in out_{S^\flat_v}(q) \ \wedge \ q(\lambda) \overset{e(\lambda)}{\to}_v r(\lambda) \ \wedge \ \forall i \in \mathbb{N} \cdot (vi \in t \Rightarrow q{\downarrow_i} \overset{e{\downarrow_i}}{\hookrightarrow}_{vi} r{\downarrow_i})$$

If v is a leaf of t, then $t{\downarrow_v} = \{\lambda\}$ and the lemma follows. Now, consider a non-leaf node v of t, and assume that the lemma holds for every child of v in t. We get that for every $i \in \mathbb{N}$ with $vi \in t$,

$$\begin{aligned} q{\downarrow_i} \overset{e{\downarrow_i}}{\hookrightarrow}_{vi} r{\downarrow_i} \quad &\Leftrightarrow \quad e{\downarrow_i} \in out_{S^\flat_{vi}}(q{\downarrow_i}) \ \wedge \ \forall u \in t{\downarrow_{vi}} \cdot (q{\downarrow_i}(u) \overset{e{\downarrow_i}(u)}{\to}_{viu} r{\downarrow_i}(u)) \\ &\Leftrightarrow \quad e{\downarrow_i} \in out_{S^\flat_{vi}}(q{\downarrow_i}) \ \wedge \ \forall u \in t{\downarrow_{vi}} \cdot (q(iu) \overset{e(iu)}{\to}_{viu} r(iu)) \end{aligned}$$

Moreover, Lemma 5.7 entails that:

$$e \in out_{S^\flat_v}(q) \quad \Rightarrow \quad \forall i \in \mathbb{N} \cdot (vi \in t \Rightarrow e{\downarrow_i} \in out_{S^\flat_{vi}}(q{\downarrow_i}))$$

We arrive at:

$$q \overset{e}{\hookrightarrow}_v r \quad \Leftrightarrow \quad e \in out_{S^\flat_v}(q) \ \wedge \ q(\lambda) \overset{e(\lambda)}{\to}_v r(\lambda) \ \wedge \ \forall u \in t{\downarrow_v} \cdot (u \neq \lambda \Rightarrow q(u) \overset{e(u)}{\to}_{vu} r(u))$$

This concludes the proof of the lemma. □

We are ready to provide a simple sufficient condition, in terms of outgoing actions, for a hierarchical covering $\widehat{\mathcal{H}}$ to be an abstraction of $\mathcal{H}$ in the sense of Definition 5.2.

Proposition 5.5. *If $out_{[\![\mathcal{H}]\!]}(q) \subseteq out_{[\![\widehat{\mathcal{H}}]\!]}(\widehat{q})$ for all $q \in \widehat{q} \in \widehat{Q}^\flat_\lambda$, then $[\![\widehat{\mathcal{H}}]\!]$ is an abstraction of $[\![\mathcal{H}]\!]$.*

Proof. Recall that $[\![\widehat{\mathcal{H}}]\!]$ is $\widehat{S}^\flat_\lambda$, and similarly $[\![\mathcal{H}]\!]$ is $S^\flat_\lambda$. Let $\widehat{q} \in \widehat{Q}^\flat_\lambda$ and $q \in \widehat{q}$, and suppose that $out_{S^\flat_\lambda}(q) \subseteq out_{\widehat{S}^\flat_\lambda}(\widehat{q})$. Pick a transition $q \overset{e}{\hookrightarrow}_\lambda r$ in $S^\flat_\lambda$. Since $\widehat{Q}^\flat_\lambda$ is a cover of $Q^\flat_\lambda$, there exists $\widehat{r} \in \widehat{Q}^\flat_\lambda$ such that $r \in \widehat{r}$, meaning that $r(u) \in \widehat{r}(u)$ for all $u \in t$. For every $u \in t$, since $\widehat{S}_u$ is a cover abstraction of S_u, it holds that $(q(u) \overset{e(u)}{\to}_u r(u)) \Rightarrow (\widehat{q}(u) \overset{e(u)}{\to}_u \widehat{r}(u))$. It follows from Lemma 5.8 that

$$\begin{aligned} q \overset{e}{\hookrightarrow}_\lambda r \quad &\Rightarrow \quad e \in out_{S^\flat_\lambda}(q) \ \wedge \ \forall u \in t \cdot (q(u) \overset{e(u)}{\to}_u r(u)) \\ &\Rightarrow \quad e \in out_{\widehat{S}^\flat_\lambda}(\widehat{q}) \ \wedge \ \forall u \in t \cdot (\widehat{q}(u) \overset{e(u)}{\to}_u \widehat{r}(u)) \\ &\Rightarrow \quad \widehat{q} \overset{e}{\hookrightarrow}_\lambda \widehat{r} \end{aligned}$$

Furthermore, it is readily seen that $q \in I^\flat_v \Rightarrow \widehat{q} \in \widehat{I}^\flat_v$ and $q \in F^\flat_v \Rightarrow \widehat{q} \in \widehat{F}^\flat_v$. Therefore the membership relation is a simulation relation from $S^\flat_\lambda$ to $\widehat{S}^\flat_\lambda$. □

The converse of Proposition 5.5 doesn't hold in general. Consider for instance the hierarchical transition system $\mathcal{H}$ given in Figure 5.10(a) and Figure 5.10(b), and the hierarchical covering $[\![\widehat{\mathcal{H}}]\!]$ induced by the cover $\{\widehat{q}_{13}, \widehat{q}_1, \widehat{q}_2, \widehat{q}_3, \widehat{q}_4\}$ where $\widehat{q}_{13} = \{q_1, q_3\}$ and $\widehat{q}_i = \{q_i\}$ for i in $\{1, 2, 3, 4\}$. Obviously, $[\![\widehat{\mathcal{H}}]\!]$ is an abstraction of $[\![\mathcal{H}]\!]$, through the simulation relation $\{(q_i, \widehat{q}_i) \mid 1 \leq i \leq 4\}$. Because of the priority $a \prec b$, $out_{[\![\widehat{\mathcal{H}}]\!]}(\widehat{q}_{13}) = \{b\}$. Hence, $out_{[\![\widehat{\mathcal{H}}]\!]}(\widehat{q}_{13})$ does not contain $out_{[\![\mathcal{H}]\!]}(q_1) = \{a\}$, even though $q_1 \in \widehat{q}_{13}$.

The condition given by the above proposition is not easy to verify as it involves quantification over all states. Even worse, it is not preserved by refinement: if an abstract state is split in two then it may be well the case that one of the smaller states has smaller set of outgoing actions. In the next section we will give a simple sufficient condition that is much easier to verify and maintain.

5.3.2 Neat covers

We present one additional requirement sufficient to guarantee that a hierarchical covering is an abstraction. As the example at the beginning of the section shows, in the presence of priorities an action can prevent some other action to happen. Intuitively this means that when grouping states together in an abstract state we should look at actions that are enabled from these states.

Consider a cover abstraction $\widehat{S}$ of a (non-hierarchical) transition system S. Since a transition exists between two states of $\widehat{S}$ if it exists between some of their elements, the set of outgoing actions in $\widehat{S}$ is just $out(\widehat{q}) = \bigcup_{q \in \widehat{q}} out(q)$. In the previous section, we saw that, in a hierarchical setting with priorities, sets of outgoing actions capture the complexity of the semantics arising from priorities. It is therefore natural to consider cover abstractions that preserve sets of outgoing actions.

Remark 5.4. *In general, $\widehat{S^\flat_\lambda}$ is not a cover abstraction of $S^\flat_\lambda$, even if we require that $out_{\widehat{S^\flat_\lambda}}(\widehat{q}) = \bigcup_{q \in \widehat{q}} out_{S^\flat_\lambda}(q)$. Indeed, it may be the case that $\widehat{q} \overset{e}{\hookrightarrow}_\lambda \widehat{r}$ even though $\neg(q \overset{e}{\hookrightarrow}_\lambda r)$ for all $q \in \widehat{q}$ and $r \in \widehat{r}$.*

To illustrate this remark, consider the hierarchical transition system $\mathcal{H}$ given in Figure 5.11(a), composed of the schema $\mathcal{S}$ and the transition systems associated to its nodes: S_λ, S_0, and S_{00}. Note that the node 00 defines the priority $c \prec b$, and the node 0 defines the priority $a \prec c$. We will see that, even though $out_{\widehat{S^\flat_\lambda}}(\widehat{q}) = \bigcup_{q \in \widehat{q}} out_{S^\flat_\lambda}(q)$, the transition system $\widehat{S^\flat_\lambda}$ is not a cover abstraction of $S^\flat_\lambda$. The semantics $S^\flat_\lambda$ of S_λ is depicted in Figure 5.11(b). To simplify notation, we name the states of $Q^\flat_\lambda$ using the names of Q_{00}. Observe that the transition $q_3 \xrightarrow{a} q_6$ has been eliminated due to the priority $a \prec c$ in the 0 node, and the transition $q_3 \xrightarrow{c} q_7$ has been eliminated due to the synchronization vectors of the λ node. Now consider the partition $\widehat{Q}_{00} = \{\widehat{q}, \widehat{p}, \widehat{r}\}$ where $\widehat{q} = \{q_1, q_2, q_3\}$, $\widehat{p} = \{q_4, q_5\}$, and $\widehat{r} = \{q_6, q_7\}$. The cover abstraction $\widehat{S}_{00}$ of S_{00} is represented in Figure 5.11(a) with the dashed boxes. Now let us go back to the eliminated transitions of S_{00}. With our cover abstraction, the state $\widehat{q}$ has four outgoing transition, and more importantly, one labeled b, and one labeled c. As the priority relation of the node 00 is

$c \prec b$, the transition $\widehat{q} \xrightarrow{c} \widehat{r}$ is eliminated in $\widehat{S}^\flat_{00}$. Continuing with the flattening of this hierarchical transition system, the priority of the 0 node does not modify the transitions, and neither does the synchronization vectors of the λ node. We obtain $\widehat{S}^\flat_\lambda$ depicted in Figure 5.11(c). Notice that $out_{\widehat{S}^\flat_\lambda}(\widehat{q}) = \bigcup_{q \in \widehat{q}} out_{S^\flat_\lambda}(q)$, and the same holds for $\widehat{p}$ and $\widehat{r}$. Still, $\widehat{S}^\flat_\lambda$ is not a cover abstraction of $S^\flat_\lambda$. Indeed, $\widehat{S}^\flat_\lambda$ contains a transition $\widehat{q} \xrightarrow{a} \widehat{r}$, but this transition is not induced by a transition of $S^\flat_\lambda$. However, $\widehat{S}^\flat_\lambda$ is an abstraction of $S^\flat_\lambda$ in the sense of Definition 5.2, which is consistent with Proposition 5.5.

Definition 5.7. *Given a transition system $S = \langle Q, \Sigma, \to, I, F\rangle$, a cover abstraction $\widehat{S} = \langle \widehat{Q}, \Sigma, \to, \widehat{I}, \widehat{F}\rangle$ of S is* neat *if for every $\widehat{q} \in \widehat{Q}$ and $q_1, q_2 \in \widehat{q}$, we have $out(q_1) = out(q_2)$. A hierarchical covering is* neat *if it is a neat cover component-wise.*

Put differently, $\widehat{S}$ is neat if $out(\widehat{q}) = out(q)$ for all $q \in \widehat{q} \in \widehat{Q}$. The notion of neat cover abstraction extends to the hierarchical setting as expected. Recall that a hierarchical covering $\widehat{\mathcal{H}}$ of a hierarchical transition system $\mathcal{H}$ is obtained from $\mathcal{H}$ by replacing each local transition system S_v by a cover abstraction $\widehat{S}_v$ of S_v. So $\widehat{\mathcal{H}}$ is neat when each $\widehat{S}_v$ is a neat cover of S_v. Our goal is to show that $[\![\widehat{\mathcal{H}}]\!]$ is a cover abstraction of $[\![\mathcal{H}]\!]$ when $\widehat{\mathcal{H}}$ is neat.

The following proposition shows that there is a strong relationship between the semantics of $\mathcal{H}$ and $\widehat{\mathcal{H}}$ when the latter is neat. In particular, as shown in Corollary 5.3, the semantics of a neat hierarchical covering is a neat cover abstraction.

Proposition 5.6. *If $\widehat{\mathcal{H}}$ is a neat hierarchical covering of $\mathcal{H}$, then for every $v \in t$, $\widehat{q}, \widehat{r} \in \widehat{Q}^\flat_v$, $q \in \widehat{q}$, and $r \in \widehat{r}$ we have:*

- $out_{S^\flat_v}(q) = out_{\widehat{S}^\flat_v}(\widehat{q})$;
- *for every $e \in \Sigma^\flat_v$: $q \overset{e}{\hookrightarrow}_v r$ iff $\widehat{q} \overset{e}{\hookrightarrow}_v \widehat{r}$ and $\forall u \in t{\downarrow_v} \cdot (q(u) \overset{e(u)}{\to}_{vu} r(u))$.*

Proof. Consider the hierarchical transition system $\mathcal{K}$ with the same hierarchical schema as $\mathcal{H}$ and $\widehat{\mathcal{H}}$, but where the local transition system of each node v is the disjoint union of those of $\mathcal{H}$ and $\widehat{\mathcal{H}}$. Since $\widehat{\mathcal{H}}$ is neat, we obtain from Corollary 5.2, applied on $\mathcal{K}$, that $out_{S^\flat_v}(q) = out_{\widehat{S}^\flat_v}(\widehat{q})$. Moreover, according to Lemma 5.8, the two following equivalences hold:

$$q \overset{e}{\hookrightarrow}_v r \quad \Leftrightarrow \quad e \in out_{S^\flat_v}(q) \ \wedge \ \forall u \in t{\downarrow_v} \cdot (q(u) \overset{e(u)}{\to}_{vu} r(u))$$

$$\widehat{q} \overset{e}{\hookrightarrow}_v \widehat{r} \quad \Leftrightarrow \quad e \in out_{\widehat{S}^\flat_v}(\widehat{q}) \ \wedge \ \forall u \in t{\downarrow_v} \cdot (\widehat{q}(u) \overset{e(u)}{\to}_{vu} \widehat{r}(u))$$

The observation that $q(u) \overset{e(u)}{\to}_{vu} r(u)$ entails $\widehat{q}(u) \overset{e(u)}{\to}_{vu} \widehat{r}(u)$ concludes the proof. □

We are now ready to show the desired result

Corollary 5.3. *If $\widehat{\mathcal{H}}$ is a neat hierarchical covering of $\mathcal{H}$, then $[\![\widehat{\mathcal{H}}]\!]$ is a neat cover abstraction of $[\![\mathcal{H}]\!]$.*

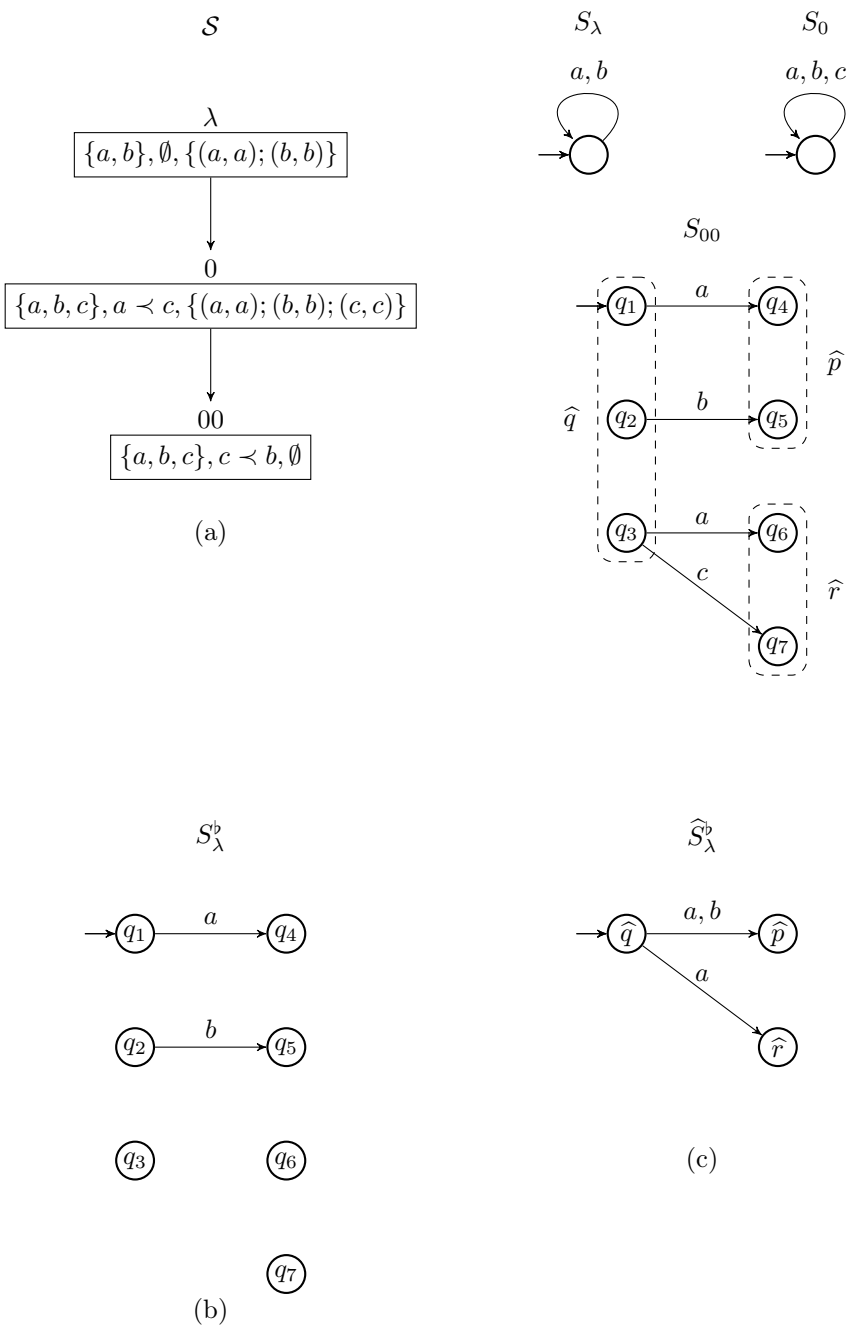


Figure 5.11: A hierarchical transition system that illustrate Remark 5.4, (a) The hierarchical schema $\mathcal{S}$, (b) the semantics $S^\flat_\lambda$ of S_λ, (c) the semantics $\widehat{S}^\flat_\lambda$ of S_λ when S_{00} is replaced by the cover abstraction $\widehat{S}_{00}$ induced by the partition $\{\widehat{q},\widehat{p},\widehat{r}\}$.

Proof. Recall that $\widehat{Q}^\flat_\lambda$ is a cover of $Q^\flat_\lambda$. The proof that the initial and final abstract states of $[\![\widehat{\mathcal{H}}]\!]$ are those of the cover abstraction induced by $\widehat{Q}^\flat_\lambda$ is the same as in Lemma 5.5. It remains to show that the transition relation $\overset{e}{\hookrightarrow}_\lambda$ of $[\![\widehat{\mathcal{H}}]\!]$ satisfies $\widehat{q} \overset{e}{\hookrightarrow}_\lambda \widehat{r}$ if and only if $q \overset{e}{\hookrightarrow}_\lambda r$ for some $q \in \widehat{q}$ and $r \in \widehat{r}$. Consider a transition $\widehat{q} \overset{e}{\hookrightarrow}_\lambda \widehat{r}$ in $\widehat{S}^\flat_\lambda$. For every $u \in t$, we get from Lemma 5.8 that $\widehat{q}(u) \overset{e(u)}{\to}_u \widehat{r}(u)$, hence, since $\widehat{S}_u$ is a cover abstraction of S_u, there exists $q_u \in \widehat{q}(u)$ and $r_u \in \widehat{r}(u)$ such that $q_u \overset{e(u)}{\to}_u r_u$. Let $q \in Q^\flat_\lambda$ and $r \in Q^\flat_\lambda$ be such that $q(u) = q_u$ and $r(u) = r_u$ for all $u \in t$. Observe that $q \in \widehat{q}$ and $r \in \widehat{r}$. We obtain from Proposition 5.6 that $q \overset{e}{\hookrightarrow}_\lambda r$. It follows that:

$$\widehat{q} \overset{e}{\hookrightarrow}_\lambda \widehat{r} \quad \Leftrightarrow \quad \exists (q, r) \in (\widehat{q} \times \widehat{r}) \cdot q \overset{e}{\hookrightarrow}_\lambda r$$

which entails that $\widehat{S}^\flat_\lambda$ is the cover abstraction of $S^\flat_\lambda$ induced by the cover $\widehat{Q}^\flat_\lambda$. Neatness of $\widehat{S}^\flat_\lambda$ follows from Proposition 5.6. □

5.3.3 CEGAR algorithm for neat covers

By the result of the previous section neat hierarchical coverings are a suitable basis for a CEGAR algorithm. Actually we will show that the same algorithm as in the priority-free case works. For this we need to understand how to refine neat covers, and how to find if an abstract path is spurious.

It turns out that refinement is completely unproblematic for neat covers.

Lemma 5.9. *If $\widehat{\mathcal{H}}$ is a neat hierarchical covering of $\mathcal{H}$, then every refinement of $\widehat{\mathcal{H}}$ is neat.*

Proof. Given two cover abstractions $\widehat{S}_1, \widehat{S}_2$ of a transition system S, if $\widehat{S}_2$ refines $\widehat{S}_1$ then every abstract state of $\widehat{S}_2$ is contained in some abstract state of $\widehat{S}_1$. Hence, neatness of $\widehat{S}_1$ entails neatness of $\widehat{S}_2$. This property obviously carries over to hierarchical coverings. □

The other good news is that verifying feasibility of a path is as easy as in priority-free cases

Lemma 5.10. *Let $\widehat{\pi} = \widehat{q}_0, e_1, \widehat{q}_1, \dots, e_n, \widehat{q}_n$ be an abstract path in $[\![\widehat{\mathcal{H}}]\!]$. If $\widehat{\mathcal{H}}$ is neat, then $\widehat{\pi}$ is feasible in $[\![\mathcal{H}]\!]$ if and only if[3] $\widehat{\pi}(v)$ is feasible in S_v for all $v \in t$.*

Proof. Assume that $\widehat{\pi}$ is feasible in $S^\flat_\lambda$. There exists a path $q_0, e_1, q_1, \dots, e_n, q_n$ in $S^\flat_\lambda$ such that $q_i \in \widehat{q}_i$ for all $0 \leq i \leq n$. It follows from Lemma 5.8 that, for every $0 < i \leq n$ and $v \in t$, $(q_{i-1}(v) \overset{e_i(v)}{\to}_v q_i(v))$. This means that $\widehat{\pi}(v)$ is feasible in S_v for all $v \in t$.

Conversely, assume that $\widehat{\pi}(v)$ is feasible in S_v for all $v \in t$. For every $v \in t$, there exists a path $q_{v,0}, e_1(v), q_{v,1}, \dots, e_n(v), q_{v,n}$ in S_v such that $q_{v,i} \in \widehat{q}_i(v)$ for all $0 \leq i \leq n$. Let us define $q_i \in Q^\flat_v$, for $0 \leq i \leq n$, by $q_i(v) = q_{v,i}$. Observe that $q_i \in \widehat{q}_i$ for all $0 \leq i \leq n$. It follows from Proposition 5.6 that $q_{i-1} \overset{e_i}{\hookrightarrow}_\lambda q_i$ for every $0 < i \leq n$. This means that $\widehat{\pi}$ is feasible in S_λ. □

[3] Recall that $\widehat{\pi}(v) = \widehat{q}_0(v), e_1(v), \widehat{q}_1(v), \dots, e_n(v), \widehat{q}_n(v)$.

These two lemmas show that we can simply use the same algorithm as in priority-free case.

Proposition 5.7. *If the initial abstraction is neat then HierarchicalCegar algorithm in Figure 5.9 is correct and terminates.*

In conclusion, once we get an initial abstraction that is neat the CEGAR algorithm in the general case is as simple and efficient as in the priority-free case. In the next section we will see how to manipulate abstractions on systems represented symbolically.

5.4 Hierarchical abstractions in AltaRica

In this section we discuss how to represent and manipulate abstractions in AltaRica (see Chapter 2). We start by describing a method that allows us to go from an AltaRica node to a hierarchical transition system and vice versa. We also propose two practical methods to obtain a neat abstraction from a AltaRica node description. Finally we discuss possible refinement methods.

Here we restrict ourselves to AltaRica nodes whose assertion does not refer to the variables of its subnodes. This restriction simplifies the translation of an AltaRica node into a hierarchical transition system. When an assertion of a node constrains variables of subnodes, then the assertion influences the semantics of subnodes. This feature would complexify substantially our translation method while its usefulness is limited.

5.4.1 AltaRica Nodes Viewed as Hierarchical Transition Systems

An AltaRica node is hierarchical, thus viewing an AltaRica node as a hierarchical transition system is quite natural. Recall that an AltaRica node is a $(6 + n + 1)$-tuple $N = \langle V, \Sigma, G, \preccurlyeq, \delta, I, A, N_0, \ldots, N_n \rangle$, where V is a set of variables, Σ is a set of events, G is a set of guarded transitions, $\preccurlyeq$ is a partial order over Σ that defines a priority relation, δ is a set of synchronization vectors, I is an initial condition, A is an assertion over the variables of V, and $N_0, \ldots, N_n$ are AltaRica subnodes (see Chapter 2 Section 2.2).

Our goal is to obtain a hierarchical transition system whose semantics is identical to the original AltaRica node semantics. To this end, we need to define a suitable hierarchical schema $\mathcal{S}$ together with transition systems that we attach to the nodes of $\mathcal{S}$.

Extracting a hierarchical schema $\mathcal{S}$ from an AltaRica node N is straightforward. Yet some issues need to be taken care of in order to obtain a proper hierarchical transition system: the naming of the nodes in the hierarchical schema, and defining proper synchronization vectors.

Given an AltaRica node N, we do a depth-first traversal of the AltaRica node, and we define, for each encountered AltaRica node, an associated node in the hierarchical schema $\mathcal{S}$ under construction as follows:

1. Create a node λ in $\mathcal{S}$
2. Set $v = \lambda$

3. Set $\Sigma_v = \Sigma_N$, $\preccurlyeq_v = \preccurlyeq_N$, and $\delta_v = \delta_N$
4. For each subnode N_i of N create a node vi in $\mathcal{S}$
5. For each subnode N_i of N, set $N = N_i$, $v = vi$, and go to back Step 3.

The construction method is simple, it generates a schema with the same hierarchical structure as an AltaRica node N. The first two steps create the λ node of the schema, and set it as the current node (v) to generate. The third step sets the elements of the schema node v to their corresponding counterparts of the AltaRica node N. Then successors to v are added for each subnode of the AltaRica node N. Finally, step 5 updates the schema node v and the AltaRica node N to continue the schema generation process.

To obtain a hierarchical transition system that is similar to our AltaRica node N, we need to label each node of the hierarchical schema with a suitable transition system. This is done as follows. Let $N_v = \langle V_v, \Sigma_v, G_v, \preccurlyeq_v, \delta_v, I_v, A_v, N_0, \ldots, N_n \rangle$ be the AltaRica node that gave rise to the node v in $\mathcal{S}$. The transition system S_v that labels v is defined as the semantics $S_v = [\![\tilde{N}_v]\!]$ of the *detached* AltaRica node $\tilde{N}_v$ given by:

$$\tilde{N}_v = \langle V_v, \Sigma_v, G_v, =, \emptyset, I_v, A_v \rangle.$$

In other words, we isolate the AltaRica node by removing its subnodes, and synchronization vectors. We also remove its priority relation in order to maintain all possible transitions at the local level, useless transition (those that will be eliminated due to the priority relation) will be eliminated when computing the semantics of the hierarchical transition system.

Example

To illustrate the translation method we just proposed, let us apply it to the AltaRica running example of Chapter 2: the AltaRica node description of a Stack of three cells given in Figure 2.13(a).

The hierarchical schema $\mathcal{S}$ that we associate to our Stack3 AltaRica node is built as follows. We create a λ node and set Σ_λ to $\Sigma_{Stack3} \cup \{\varepsilon\}$, and $\preccurlyeq_\lambda$ to $\preccurlyeq_{Stack3}$ (steps 1 through 3). The nodes 0 and 1 are added to $\mathcal{S}$ as the successors of λ and represent respectively the AltaRica sub nodes of Stack3, Stack1, and Stack2 (step 4). The synchronization vectors of the λ node are created with respect to the synchronization vectors of Stack3. For instance $\langle pushT, Top.push \rangle$ becomes $(pushT, push, \varepsilon)$. The current AltaRica node becomes Stack1, and the node of $\mathcal{S}$ becomes 0 and the procedure goes back to step 3. Once done we obtain the resulting hierarchical schema $\mathcal{S}$ given in Figure 5.12(a). The nodes $\lambda, 0, 1, 10, 11$ represent respectively the AltaRica nodes: Stack3, Stack1, Stack2, Stack1, and Stack1.

To finish the construction of our hierarchical transition system we need to associate to each node of $\mathcal{S}$ a transition system. Note that when isolated as presented above the AltaRica nodes Stack3 and Stack2 describe an identical transition system S_a that is given in Figure 5.12(b). This transition system will label the nodes λ and 1. The AltaRica node

Stack1 describes the transition system S_b and is given Figure 5.12(c). This transition system labels the nodes $0, 10$, and 11. This assignment is given in Figure 5.12(d).

We have seen how we can obtain an hierarchical transition system from an AltaRica node. We now turn our attention to abstraction methods for AltaRica nodes.

5.4.2 Abstracting a Leaf AltaRica Node

Predicate abstraction [GS97] is a particular instance of cover abstraction, where the cover is the partition induced by a finite collection of subsets of the state space. In practice, the partition is not constructed explicitly. Instead, each equivalence class is represented by a bit vector, and, accordingly, predicate abstractions are constructed and explored symbolically. In this section, we show how to compute neat predicate abstractions of AltaRica nodes. Encoding these abstractions by AltaRica nodes themselves isn't straightforward, since assignments in AltaRica are deterministic. So we first present predicate abstractions in terms of boolean transition systems, and then we discuss their representation by AltaRica nodes.

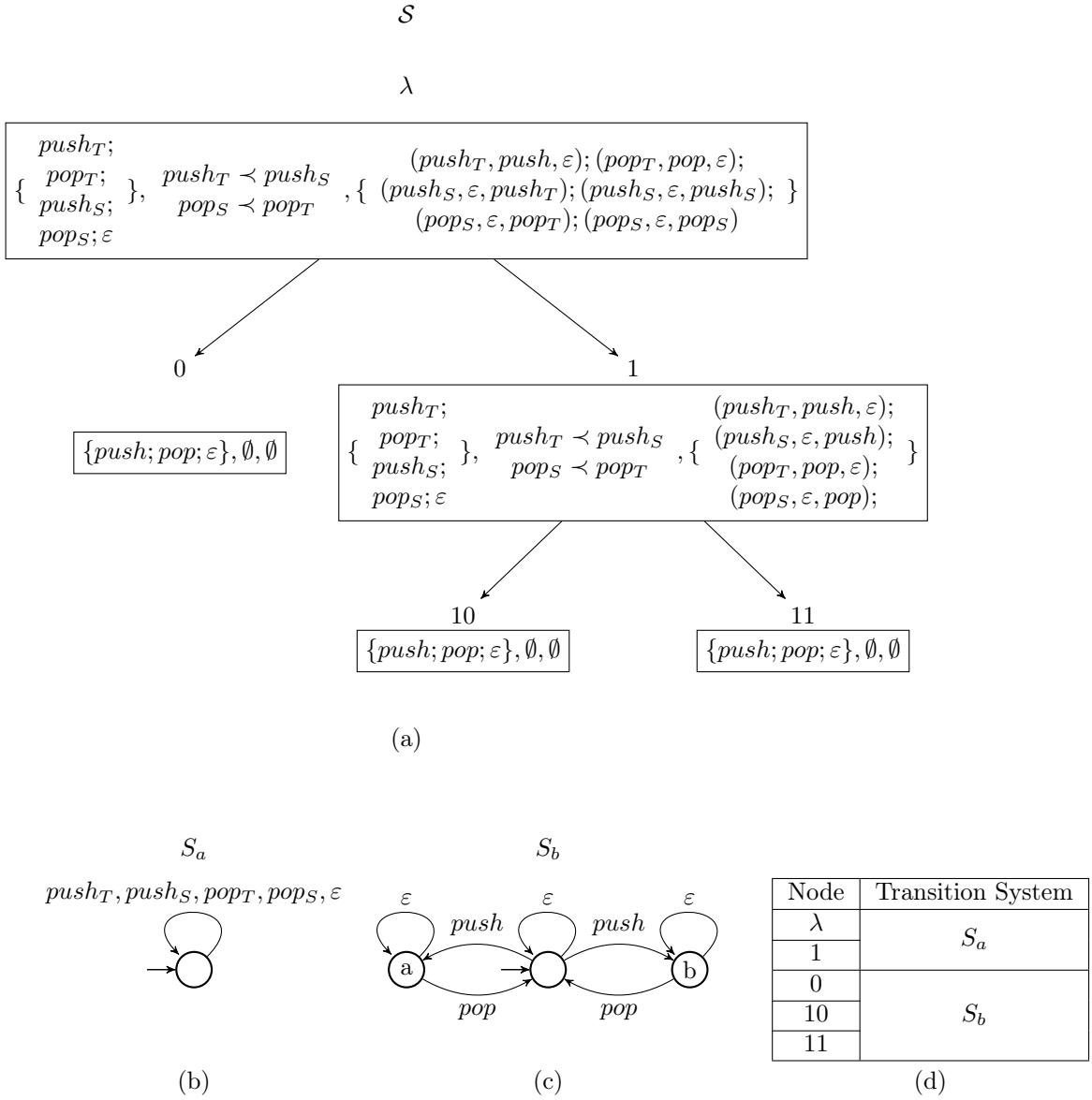
Consider a leaf AltaRica node $N = \langle V, \Sigma, G, =, \emptyset, \mathcal{A}, I\rangle$ and a finite set P of first-order formulas over the variables V of N. Recall that the semantics of the AltaRica node N (see Chapter 2.2.1) is given by the transition system $[\![N]\!] = \langle Q, \Sigma, \rightarrow, I\rangle$ where each state of Q is a configuration of the AltaRica node N. Each predicate $p \in P$ defines a subset of Q, namely the set $\{q \in Q \mid q \models p\}$. Therefore, the set of predicates P induces, in a natural way, a partition of the state space Q. This partition corresponds to the equivalence relation $\equiv$ on Q defined by $q \equiv q'$ if q and q' satisfy the same predicates of P. As mentioned before, the predicate abstraction of N with predicates P is nothing more than the cover abstraction of $[\![N]\!]$ induced by this partition. Observe that the number of states of the predicate abstraction is, in the worst case, exponential in the size of P. Therefore, predicate abstractions must be computed (and explored) in a symbolic way.

Following the classical approach of [GS97], we introduce a (fresh) boolean variable $\mathtt{b}_p$ for each predicate $p \in P$. Intuitively, each $\mathtt{b}_p$ represents the truth value of p (i.e., $\mathtt{b}_p$ is `true` when p holds, and is `false` otherwise). Let B_P denote the set $B_P = \{\mathtt{b}_p \mid p \in P\}$. An element of the above-mentioned partition induced by P is, therefore, a valuation $\vec{b}$ of the variables in B_P (i.e., a function from B_P to $\{\mathtt{true}, \mathtt{false}\}$). The abstraction of a concrete configuration $\vec{v}$ is the valuation $\vec{b}$ that maps each $\mathtt{b}_p$ to the truth value of p in $\vec{v}$. This abstraction relationship is expressed by the following first-order formula:

$$\alpha(\vec{\mathtt{v}}, \vec{\mathtt{b}}) \triangleq \bigwedge_{p \in P} \mathtt{b}_p \Leftrightarrow p(\vec{\mathtt{v}})$$

Recall that covers of Q may not contain the empty set. Correspondingly, we restrict the state space of the predicate abstraction to those valuations that are the abstraction of some concrete configuration. This condition is formally expressed by the first-order formula $\mathcal{A}_P$ defined by:

$$\mathcal{A}_P(\vec{\mathtt{b}}) \triangleq \exists \vec{\mathtt{v}} \cdot (\mathcal{A}(\vec{\mathtt{v}}) \wedge \alpha(\vec{\mathtt{v}}, \vec{\mathtt{b}}))$$



Node	Transition System
λ	S_a
1	
0	S_b
10	
11	

Figure 5.12: An hierarchical transition system that models a stack, (a) The hierarchical schema $\mathcal{S}$, (b) the transition systems S_a links two stacks, (c) the transition system S_b that models a stack that can contain an object a or an object b, (d) the mapping table that matches each node of t to a transition system.

Similarly, initial abstract states are the valuations that are the abstraction of some concrete initial configuration. This condition is formally expressed by the first-order formula I_P defined by:

$$I_P(\vec{\mathtt{b}}) \triangleq \exists \vec{\mathtt{v}} \cdot (\mathcal{A}(\vec{\mathtt{v}}) \wedge I(\vec{\mathtt{v}}) \wedge \alpha(\vec{\mathtt{v}}, \vec{\mathtt{b}}))$$

To complete the symbolic representation of the predicate abstraction, we express its labeled transition relation by the first-order formula T_P as follows:

$$T_P(\vec{\mathtt{b}}, e, \vec{\mathtt{b}}') \triangleq \bigvee_{(g,e,u)\in G} \exists \vec{\mathtt{v}}\, \exists \vec{\mathtt{v}}' \cdot (\mathcal{A}(\vec{\mathtt{v}}) \wedge \mathcal{A}(\vec{\mathtt{v}}') \wedge g(\vec{\mathtt{v}}) \wedge \vec{\mathtt{v}}' = u(\vec{\mathtt{v}}) \wedge \alpha(\vec{\mathtt{v}}, \vec{\mathtt{b}}) \wedge \alpha(\vec{\mathtt{v}}', \vec{\mathtt{b}}'))$$

Put differently, given two valuations $\vec{b}, \vec{b}' : B_P \to \{\mathtt{true}, \mathtt{false}\}$ and an event e of Σ, the triple $(\vec{b}, e, \vec{b}')$ satisfies the formula T_P if and only if there exists an AltaRica transition (g, e, u) of G and two valuations $\vec{v}, \vec{v}' : V \to \mathcal{D}(V)$ that satisfy following conditions:

- $\vec{v}$ and $\vec{v}'$ are configurations of N,
- $\vec{v}$ satisfies the guard g,
- $\vec{v}'$ satisfies the post condition of the transition w.r.t. $\vec{v}$, and
- $\vec{v}$ and $\vec{v}'$ are abstracted by $\vec{b}$ and $\vec{b}'$, respectively.

The quintuple $N_P = \langle B_P, \Sigma, T_P, \mathcal{A}_P, I_P \rangle$ is what we meant previously by symbolic representation of the predicate abstraction. Indeed, the obvious labeled transition system providing the operational semantics of N_P is readily seen to be "isomorphic" to the predicate abstraction of N with predicates P. Note that N_P is not an AltaRica node per se, since the formula T_P expressing the transition relation is not deterministic. In order to use existing model-checking tools for AltaRica, it is desirable to have abstraction techniques that produce abstractions expressible in AltaRica. So we now address this issue and present two methods for encoding symbolic predicate abstractions as AltaRica nodes.

Transition Decomposition

The first method decomposes the transition relation T_P so that we only get deterministic assignments. To this end, we iterate over all possible updates of the boolean variables B_P, and construct AltaRica transitions for each update. Formally, the AltaRica node produced by the transition decomposition method is the quintuple $N_P^{Trans} = \langle B_P, \Sigma, G_P^{Trans}, =, \emptyset, \mathcal{A}_P, I_P \rangle$ where all variables are state variables and the set of AltaRica transitions G_P^{Trans} is defined by:

$$G_P^{Trans} \triangleq \bigcup_{\vec{b}' : B_P \to \{\mathtt{true},\mathtt{false}\}} \{(T_P(\vec{\mathtt{b}}, e, \vec{\mathtt{b}}') \wedge \vec{\mathtt{b}}' = \vec{b}', e, \vec{\mathtt{b}} := \vec{b}') \mid e \in \Sigma\}$$

Unfortunately, this method leads to an exponential blow-up in the number of AltaRica transitions.

Flow Decomposition

The second method overcomes the limitation of deterministic assignments with the help of flow variables. Here, the boolean variables B_P will be flow variables, and the assertion of the constructed AltaRica node will guarantee that changes of these flow variables are legitimate. Formally, the AltaRica node produced by the flow decomposition method is the quintuple $N_P^{Flow} = \langle B_P \cup \{\mathtt{a}_p \mid p \in P\} \cup \{\mathtt{last}\}, \Sigma, G_P^{Flow}, =, \emptyset, \mathcal{A}_P^{Flow}, I_P \rangle$ where B_P are flow variables and all other variables are state variables. The $\mathtt{a}_p$ and $\mathtt{b}_p$ variables are boolean, and `last` ranges over the set of events Σ. The $\mathtt{a}_p$ variables are used to maintain the previous value of the $\mathtt{b}_p$ variables, and the variable `last` keeps track of the event labeling the last fired transition. With these variables, we can use the assertion to express the transition relation T_P. The AltaRica transitions G_P^{Flow} and assertion $\mathcal{A}_P^{Flow}$ are defined as follows:

$$\begin{array}{lcl} G_P^{Flow} & \triangleq & \{(\mathtt{true}, e, \vec{\mathtt{a}} := \vec{\mathtt{b}}, \mathtt{last} := e) \mid e \in \Sigma\} \\ \mathcal{A}_P^{Flow} & \triangleq & \bigwedge_{e \in \Sigma} \mathtt{last} = e \Rightarrow T_P(\vec{\mathtt{a}}, e, \vec{\mathtt{b}}) \end{array}$$

Remark that the additional variables $\mathtt{a}_p$ and `last` are not constrained by the initial condition I_P. Compared to the first method, the flow decomposition method is more succinct. However, the semantics $[\![N_P^{Flow}]\!]$ is not isomorphic to the predicate abstraction anymore. This is due to the additional variables that lead to a duplication of the abstract states. This kind of redundancy cannot be captured by our cover abstraction formalism, but the latter could be easily generalized to manage such duplications.

Elimination of Existential Quantifications

In the transition decomposition and flow decomposition methods, existential quantifiers may appear in the guard and the assertion of a node due to the use of the formulas $T_P, \mathcal{A}_P$, and I_p. The AltaRica language does not support quantifiers and, therefore, we need to eliminate them. This task can be performed in various ways:

- With the help of a SAT solver, we can compute all boolean valuations satisfying the existentially quantified formula, and compute an equivalent propositional formula,
- When the formula falls in a class that admits quantifier elimination, such as Presburger arithmetic, we can simply use an external dedicated tool for this task.

5.4.3 Neat Covers

We now define a set of predicates that will allow us to generate a neat cover abstraction of an AltaRica node using our transition decomposition and flow decomposition abstraction methods.

The abstraction we intend to use, is an AltaRica node obtained by the transition, or flow decomposition methods. These methods generate an AltaRica node based on a predicate abstraction. We therefore need to define a set of predicates whose predicate

abstraction will induce a neat cover abstraction. From its definition, it is clear that to this end, we need to characterize with our predicates the configurations that can fire a transition labeled with the same events. More formally, given an AltaRica node $N = \langle V, \Sigma, G, =, \emptyset, \mathcal{A}, I \rangle$, for each event $e \in \Sigma$ we define a predicate p_e as follows:

$$p_e(\vec{v}) = \mathcal{A} \wedge \bigvee_{(g,e,u) \in G} g \wedge [\vec{v} := u(\vec{v})]\mathcal{A}$$

Such a predicate is called *event predicate*, and the set all event predicates is written P_Σ. As the name suggest it, such a predicate says if there is an outgoing transition labeled with the associated event. This is quite natural since from Definition 5.7, neatness is defined with respect to outgoing transitions. So each event predicate p_e characterizes the configurations of the AltaRica node (that is valuations of $\mathcal{D}(V)$ that satisfy the assertions $\mathcal{A}$) that can fire a transition labeled with the event e (i.e., that satisfy the guard g, and whose update u is a configuration). With the help of these predicates obtaining an neat cover abstraction of the semantic of a AltaRica node is straightforward. Summarizing we obtain:

> *Given an AltaRica node N, and P_Σ its set of transition predicates. The predicate abstraction of $[\![N]\!]$ with predicates P_Σ is a neat cover abstraction of $[\![N]\!]$.*

It follows that with the help of event predicates, we can easily obtain a neat cover abstraction of an AltaRica node. Now equipped with a abstraction method that can give us our initial abstraction, we can turn our attention to the refinement of these abstractions.

5.4.4 Refinement of Abstract Detached AltaRica Nodes

We have seen how to translate an AltaRica hierarchical node into a hierarchical transition system, and how to abstract detached AltaRica nodes. When a hierarchical transition system is verified by the HierarchicalCegar algorithm, a refinement of one of the transition systems is required when a spurious abstract path has been detected. We now turn our attention to this refinement step, when HierarchicalCegar algorithm is used to verify an AltaRica hierarchical node.

Let us go back to the HierarchicalCegar algorithm given in Figure 5.9. In this algorithm, when a spurious abstract run $\widehat{\pi}$ is detected Line 3, with the help VerifyHierarchicalPath algorithm a transition system $\widehat{S}_u$ in the hierarchy is selected to be refined Lines 6-7. Thanks to VerifyHierarchicalPath we know that the abstract path $\widehat{\pi}(u)$ is a spurious abstract path of $\widehat{S}_u$. We therefore need to refine this neat cover abstraction.

First note that the refinement methods presented in Chapter 4.3.2 can be applied, but when dealing with an hierarchical transition system induced by a hierarchical AltaRica node, we can go one step further an delegate this step to an external tool. As we have seen previously in this section, it is possible to abstract an AltaRica node description into another AltaRica node description with the help of a set of predicates. Recall that each

AltaRica node of the hierarchy is abstracted with its own predicates. The refinement we propose here is the "usual" extension of this set of predicates P_u with one or more new predicates that eliminates $\widehat{\pi}(u)$ from $Run(\widehat{S_u})$. Extending a predicate abstraction with new predicates clearly falls into our refinement setting presented in Section 5.2.1.

Various tools and methods to discover such predicates are available (see Chapter 3.2.3), a particularly suitable approach was proposed by McMillan in [McM04] where invariants where generated with the help of Craig interpolants and a theorem prover out of a spurious abstract run. Another similar approach is the the path invariants method [BHMR07] proposed by Beyer et al. Once the new boolean predicate is discovered, we can abstract once again the AltaRica hierarchical node an resume the verification process of HierarchicalCegar. As a final remark, note that thanks to our boolean abstraction method, we can translate the abstraction of the AltaRica node directly into the formalism of external tools like NuSMV [CCG+02] and FOCI [McM04].

5.5 Concluding remarks

In this section we have considered the situation where we want to apply CEGAR algorithm to a hierarchical transition system. We wanted to do this without calculating the semantics of the hierarchical system. We have proposed to use hierarchical abstractions. This has three advantages: an abstraction is represented in a succinct way, it is easy to verify if an abstract path is spurious, the abstraction reflects the logical structure of the system.

One may ask what happens if we would like to use standard, not hierarchical, abstractions. For this we need to be able to provide an initial abstraction, and verify if an abstract path is spurious. In case of priority-free systems this would be rather easy: the same algorithm presented in Section 5.2.2 can be used to test if a path is spurious. Calculating initial abstraction is relatively simple too. In the presence of priorities the task is much more difficult. In particular, it is not clear how to efficiently calculate the initial abstraction. By definition, this should be one state system with all transitions that exist in a real system. At present, we do not know an easier way to calculate it than to essentially calculate the semantics of the system.

Chapter 6

Implementation

Introduction

The standard Cegar algorithm and our PCegar algorithm have been implemented in Mec 5.3 (Mec 5 for short). This implementation served to us to benchmark the algorithms on a set of test models. We have also applied our CEGAR algorithms on a large industrial model. This chapter is dedicated to the presentation of Mec 5, and the implementation of the CEGAR algorithms. We start in Section 6.1 by a presentation of Mec 5. The implementation of the CEGAR algorithms is the focus of Section 6.2. A detailed analysis of the benchmarks is given in Section 6.3. The same section presents an industrial model we have treated using our tools.

6.1 Mec 5

Mec 5 [Vin03, GV04] is a relation computation tool usually used as an AltaRica model checker. Mec 5 have been developed by Aymeric Vincent as part of his PhD thesis [Vin03]. Since its original release in 2003, Mec 5 have been used in many research projects. Among them, Claire Pagetti [Pag04] implemented in Mec 5 verification methods for the timed extension of AltaRica she proposed. Romain Bernard [Ber09] used Mec 5 for RAMS (Reliability Availability Mutability and Security) studies. More recently, Nicolas Aucouthurier tested a BDD interpolation based CEGAR refinement method [Auc08] in our CEGAR extension of Mec 5.

AltaRica nodes are the standard input of Mec, but a user can also define an n-array relation with the Mec specification language. This specification language, allows the user to define relations using first-order logic together with the μ-calculus least fix point (μ)

and greatest fix point (ν) operators[1]. Thanks to these operators, it is simple to perform a model checking tasks with Mec 5. Most often it is done as follows: An AltaRica node is loaded in Mec 5, and a property P is specified (in most cases, it is a unary relation over the set of configurations of the node). The AltaRica node defines a transition system model M together with the set of initial states of the model. The objective of verification is to check if P holds in all states reachable from the initial states of the model. This can be done in two ways:

Forward verification Compute all the reachable configurations from the initial states of the model.

Backward verification Compute the coreachable configurations from the "error" states (the configurations not satisfying P).

Once one of these two sets is calculated by Mec 5 a simple set intersection test permits to conclude. We now present in more details the use of Mec 5 with AltaRica nodes.

6.1.1 AltaRica Nodes & Mec 5

As we have said, Mec 5 is a relation computation tool. Relations in Mec 5 are managed with the help of a custom BDD package. We refer the interested reader, to the PhD thesis of Aymeric Vincent [Vin03] for a detailed presentation of this BDD package. Relations are the basic objects of Mec 5. They are used to represent the semantic of an AltaRica node. The semantic of an AltaRica node is obtained using the semantic composition method presented in Chapter 2 Section 2.2.2. In Mec an AltaRica node defines data types accessible to the user, among them we have the configurations, and the transition relation. The following listing illustrates the use of Mec 5:

```
[mec] :ar-load ./Stack3.alt  // Load
[mec] configurations(s : Stack3!c) := true;
        configurations: (Stack3!c) -> bool
[mec] :rel-cardinal configurations
        cardinal of configurations: 27
[mec] :rel-cardinal Stack3!t
        cardinal of Stack3!t: 91
```

In this Mec 5 session, we start by loading the AltaRica node `Stack3` of Figure 2.13(a) of Chapter 2 with the `ar-load` command. Once the node is loaded, we then define the relation *configurations* that represents the configurations of `Stack3`. Then with the help of the command `rel-cardinal` we obtain the cardinal the relation *configurations*, and the relation *Stack*3!*t* that represents the transition relation of `Stack3`. In our example we have 27 configurations and 91 transitions.

Continuing with this Mec 5 session, we can compute the reachable configurations of our `Stack3` node as follows:

[1] This logic is also known as the Park's μ-calculus.

```
[mec] PostStarOfInit(s:Stack3!c) += Stack3!init(s) |
                                    <x>( PostStarOfInit(x) & <e>(Stack3!t(x,e,s)));
      Stack3!init: (Stack3!c) -> bool
      Stack3!t: (Stack3!c, Stack3!ev, Stack3!c) -> bool
      PostStarOfInit: (Stack3!c) -> bool
[mec] :rel-cardinal PostStarOfInit
      cardinal of PostStarOfInit: 15
```

The relation $PostStarOfInit$ characterizes the reachable configurations of `Stack3` from its initial configurations (obtained with the predicate $Stack3!init(s)$). This relation is defined in a standard way with the help of the least fix point operator $+=$, and the transition relation of the AltaRica node: $Stack3!t$. The set of reachable configurations of `Stack3` is not empty, more precisely there are 15 reachable configurations. We can check a property on our `Stack3` AltaRica node. For example, we will verify (model check) that in `Stack3` the Top subnode cannot hold an element if the $Stack$ subnode is empty. To do so, we define in Mec the relation Err as follows:

```
[mec] Err(s : Stack3!c) := s.Top.object != no &
                           (s.Stack.Top.object = no & s.Stack.Stack.object = no);
      Err: (Stack3!c) -> bool
[mec] :rel-cardinal Err
      cardinal of Err: 2
[mec] ReachErr(s : Stack3!c) := PostStarOfInit(s) & Err(s);
      ReachErr: (Stack3!c) -> bool
[mec] :rel-cardinal ReachErr
      cardinal of ReachErr: 0
```

The Err relation characterizes the undesired configurations of our `Stack3` node: the subnode Top is not empty (the clause $s.Top.object! = no$), but the $Stack$ subnode is empty (the clause $s.Stack.Top.object = no \,\&\, s.Stack.Stack.object = no$). The Err relation contains two configurations: the Top subnode holding an object of type a or b, and the objects of the $Stack$ subnode set of no. The $ReachErr$ relation is defined in order to determine the configurations that belongs to the $PostStarOfInit$ and Err relations (a set intersection). Finally, the command `rel-cardinal` allows us to conclude that these undesired configurations are not reachable. Hence, the `Stack3` node satisfies the property.

Now that we have seen Mec 5 in use, we will briefly go over its implementation, and describe our CEGAR extension.

6.1.2 Mec 5 & CEGAR

Mec 5 is written in C. The code is organized in modules that manage each of its data types or functions. The basic component of Mec 5 is the relation type. Relations are managed in Mec 5 with the help of an custom BDD module. AltaRica nodes for instance are defined with the help of relations. For a given AltaRica node, three relations are defined: the set of configurations, the set of events, and the transition relation. Once these relations are defined, Mec 5 allows us to compute user defined expressions over these relations.

Another point of interest of Mec 5 is its memory management: a garbage collector is implemented within Mec 5. This is classical in based BDD tools since large amount of memory can be allocated and freed often. The use of a garbage collector minimizes system calls, and speeds the application. On the other hand, garbage collectors may maintain unused memory (until a certain threshold is reached) and artificially increase memory need.

As all BDD tools Mec 5.2 (the latest release before its extension with our CEGAR methods) may of course not terminate on large models: it could be that the representation of the set of reachable configurations of the model is simply too big. In this case, while computing $PostStarOfInit$ relation from our running example MEC would report an insufficient memory message. This is a well-known BDD-blowup problem (see [Vin03] for examples). Indeed, the number of nodes of a BDD can grow exponentially, due to the conjunction and disjunction operation occurring during the computation of the fix point.

Our CEGAR extension of Mec 5 is intended to mitigate this blowup problem. The CEGAR algorithms are build upon the preexisting modules of Mec 5 (BDDs, AltaRica...). The algorithms take advantage of the concise representation offered by the BDDs, while avoiding (as much as possible) the computation of the reachable configurations of the model under analysis.

The CEGAR Extension

The CEGAR extension implemented in Mec 5 is composed of about 4000 lines of code. Its modular decomposition permits a simple and quick extension of the algorithm in a "plugin" way. Naturally, the decomposition of the data structure: abstractions, and abstract counterexamples, and its functional aspect: abstract counterexample search methods, abstract counterexample verification methods, and abstraction refinement methods follow the CEGAR scheme (see Figure 3.2).

This decomposition allowed us to implement the Cegar and PCegar algorithms of Chapter 4 in a generic way: the CEGAR loop is a single method parameterized with the pruning (and certified pairs inference) steps to perform. The method implementing the loop calls the different modules implementing the functional aspect of the CEGAR loop. Each module defines a clear input/output interface that must be implemented by the functions performing the task. The various functions implementing a module are registered and can be selected by the CEGAR loop wrt user specified options. This clear decomposition and interface definition ease the extension of the CEGAR algorithms. We now present the different steps of our CEGAR algorithm for the verification of AltaRica nodes.

Abstractions. An abstraction of an AltaRica node is represented with the help of an explicit transition system: Unlike the configurations and the transition relation of an AltaRica node that are represented as relations, the set of states and transitions are implemented as a collection of objects. The set of states is induced by the abstraction method: Either a boolean predicate abstraction (see Chapter 3 Section 3.2.1) or a cover

abstraction (see Chapter 4 Section 4.1.1). Each state of the abstraction is associated to a Mec 5 relation that represents a set of configurations (given by the abstraction method). The transition relation is the classical existential $\exists\exists$ transition relation induced by the abstract state space (see Chapter 4 Section 4.1.1). Once computed the abstraction is used by the counterexample search method.

CounterExamples Verification. A counterexample is a sequence of states and transitions that forms a path in the abstraction. The feasibility verification of an abstract counterexample extends this path with the reachable (or coreachable) configurations computed during the verification of the path. If proven spurious, the counterexample together with the abstraction are sent to the refinement method.

Abstraction Refinement. The goal of the abstraction refinement methods is to eliminate the spurious counterexample found be the counterexample search method. Classically, the refinement methods modify the set of states of the abstraction. In our implementation, once the set of states modified, the refinement methods return to a transition relation refinement method the "removed" state, and the new ones.

Transition Relation Refinement. The refinement of the transition relation is performed as follows: The incoming and outgoing transitions of the eliminated state, are redistributed over the new states. This optimization allows us to avoid useless computations of a new transition relation for our abstraction.

6.2 The CEGAR Implementation in Mec 5

First, we have extended Mec with the classical CEGAR framework as presented in Section 3.2. Our implementation of the CEGAR framework allows to compute an abstraction of a AltaRica model for a given safety property. Then, the CEGAR loop is implemented classically: extraction of an abstract counterexample, verification of the abstract counterexample, and if proven spurious the abstraction is refined to eliminate the abstract counterexample.

6.2.1 Abstraction

For a given AltaRica model, a set of initial states, a safety property, and a set of predicates, an initial abstraction is computed using the command `ar-cegar-init`. The syntax of the command is:

`:ar-cegar-init` <AltaRica Node> <Initial States> <Error States> $[P_1, \ldots, P_n]$

The arguments are:

Argument	Description
AltaRica Node	Name of the AltaRica node to analyze.
Initial States	A predicate that defines the set of concrete initial states to consider.
Error States	A predicate that defines the set of concrete states that violate the safety property.
$P_1, \ldots, P_n$	A list of predicates.

The command produces either a boolean predicate abstraction (see Section 3.2.1), or a cover abstraction (see Section 4.1.1).

When set to generate a boolean predicate abstraction, the `ar-cegar-init` computes the set of abstract states as a partition of the concrete state space. As in [GS97] the predicates $P_1, \ldots, P_n$ are used to induce equivalence classes over the AltaRica node configurations: two configurations are abstracted by the same abstract state if they cannot be distinguished w.r.t. the input predicates (i.e. they satisfy the same set of predicates).

When set to generate a cover abstraction, the `ar-cegar-init` command computes the set of abstract states as follows: for every predicate P_i an abstract state is defined which represents all configurations satisfying the predicate. Additionally, if the union of all abstract states does not cover the AltaRica node configurations, a new abstract state is added and is defined as the complement of all previously defined abstract states.

By default, a boolean predicate abstraction is computed by the `ar-cegar-init` command. To generate a cover abstraction the option: `cegar-use-cover-abstraction` must be set prior to any call of the `ar-cegar-init` command.

```
:set cegar-use-cover-abstraction
:ar-cegar-init Node Init Error P_1, ..., P_n
```

The transition relation of the abstraction is a classical existential abstract transition relation.

Mec 5 defines two transition relations: a standard transition relation, and a "super" transition relation. The super transition relation does not take into account the node assertion. Yet it is sound to manipulate the super transition relation as long as the AltaRica node does not use priorities. For more details about the super transition relation the reader is referred to Mec 5 documentation [Mec10].

By default, the `ar-cegar-init` command will use the classical transition relation to compute the abstract transition relation. In order to use the super transition relation instead, the option `cegar-use-super-transition` must be set prior to any call of the `ar-cegar-init` command.

```
:set cegar-use-super-transition
:ar-cegar-init Node Init Error P_1, ..., P_n
```

A method is also available to automatically generate predicates from an AltaRica node. This generates all the predicates that occur in the guards of transitions of the AltaRica node. For moderately complicated models this can give of an order of a hundred predicates. Note that when `ar-cegar-init` command computes a boolean pred-

icate abstraction with n predicates, there is up to 2^n abstract states that are generated. Hence, if n is large it is preferable to use cover abstractions. To use this option `cegar-use-guards-as-predicates` must be set prior to any call of the `ar-cegar-init` command.

```
:set cegar-use-guards-as-predicates
:ar-cegar-init Node Init Error
```

6.2.2 Counterexample extraction

At each iteration of the CEGAR loop, an abstract counterexample is extracted. This counterexample is extracted from the abstraction using a graph search algorithm. We have implemented in our CEGAR framework two classical graph search algorithms: breadth first search (BFS) and depth first search (DFS). To select a counterexample search algorithm the option `cegar-search-algorithm` must be set. The default value is `BFS`, to use DFS instead the option must be set to `DFS` as follows:

```
:set cegar-search-algorithm DFS
```

6.2.3 Counterexample analysis

Once an abstract counterexample have been selected, it has to be analyzed in order to determine if it is spurious or not. The forward analysis algorithm VerifyPath (see Chapter 3 Section 3.2.2) have been implemented to this end. The pseudo code of the implemented algorithm is given in Figure 6.1. A dual algorithm, that performs a backward analysis of the abstract counterexample: from the final state back to the initial state has also been implemented (the pseudo code is given in Figure 6.2). In our implementation, these algorithms return the position of the failure state within the abstract counterexample, as well as the set of concrete reachable (resp. coreachable) states for each abstract state from the initial (resp. final) abstract state to the failure state when using the forward (resp. backward) counterexample analysis algorithm.

CE-Forward-analysis $(S, \widehat{\pi})$

Input: A transition system S, an abstract counterexample π.

```
1  T[0] = q̂_0 ∩ I
2  i = 1
3  while i ≤ |π| ∧ X ≠ ∅ do
4      i = i + 1
5      T[i] = post(T[i − 1]) ∩ q̂_i
6  done
7  if i = |π|
8      return (−1,T)
9  else
10     return (i,T)
```

Figure 6.1: The forward counterexample analysis pseudo code.

CE-Forward-analysis $(S, \widehat{\pi})$

Input: A transition system S, an abstract counterexample π.

```
1  T[n] = q̂_n ∩ F
2  i = n
3  while i > 0 ∧ X ≠ ∅ do
4      i = i − 1
5      T[i] = pre(T[i + 1]) ∩ q̂_i
6  done
7  if i = 0
8      return (−1,T)
9  else
10     return (i,T)
```

Figure 6.2: The backward counterexample analysis pseudo code.

To select a counterexample analysis algorithm the option `cegar-ce-analysis` must be set. The default value is `FORWARD`, to use backward algorithm instead the option must be set to `BACKWARD` as follows:

```
:set cegar-ce-analysis BACKWARD
```

6.2.4 Abstraction Refinement Heuristics

Once a counterexample is exhibited, the user can automatically refine the current abstraction. The current release of the CEGAR framework implemented in Mec 5 proposes two refinement heuristics: the *direct*, and *sigma* heuristics. These heuristics split the failure state identified by the counterexample analysis algorithm into two new abstract states. The new abstract states form a partition of the failure state.

Abstract States Refinement Heuristics

In order to present the refinement heuristics, we consider a spurious abstract counterexample $\widehat{\pi} = \widehat{q}_0, \widehat{q}_1, \ldots, \widehat{q}_n$. The failure state of this counterexample is $\widehat{q}_i$, and the set of reachable (resp. coreachable) concrete states is denoted by F (resp. B). Hence, our forward (resp. backward) analysis algorithm returned (i, F) (resp. (i, B)) when analyzing $\widehat{\pi}$. We will also denote by $\widehat{d}$ the dead end concrete states of the failure state. Hence we have $\widehat{d} = F[i]$ if $\widehat{\pi}$ was analyzed using the forward counterexample algorithm, or $\widehat{d} = B[i]$ if $\widehat{\pi}$ was analyzed using the backward counterexample algorithm.

The *direct* refinement heuristic splits the failure state $\widehat{q}_i$ into two new abstract states: $\widehat{q}'$ and $\widehat{q}''$ such that $\widehat{q}' = \widehat{d}$ and $\widehat{q}'' = \widehat{q}_i \setminus \widehat{q}'$. The abstract state space is then refined by eliminating $\widehat{q}_i$ and adding $\widehat{q}'$ and $\widehat{q}''$ to $\widehat{Q}$ the set of abstract states. Observe that the resulting abstraction is dependent on counterexample analysis algorithm. Indeed, the set of dead end concrete states differs if the counterexample analysis algorithm used performs a forward analysis or a backward analysis. Hence, the direct refinement heuristic will generate different refined abstraction depending on the abstract

counterexample analysis algorithm used. To use direct refinement algorithm the option `cegar-abstraction-refinement-algorithm` must be set to `DIRECT` as follows:

```
:set cegar-abstraction-refinement-algorithm DIRECT
```

The *sigma* refinement heuristic splits the failure state $\widehat{q}_i$ into two new abstract states: $\widehat{q}'$ and $\widehat{q}''$. As for the direct refinement heuristic, the computed abstraction depends on the abstract counterexample analysis algorithm. If the abstract counterexample was analyzed using the forward algorithm the new abstract states are defined as follows: $\widehat{q}' = pre(\widehat{q}_{i+1}) \cap \widehat{q}_i$, and $\widehat{q}'' = \widehat{q}_i \setminus \widehat{q}'$. If the backward analysis algorithm was used, the new abstract states are computed as: $\widehat{q}' = post(\widehat{q}_{i-1}) \cap \widehat{q}_i$, and $\widehat{q}'' = \widehat{q}_i \setminus \widehat{q}'$.

To use sigma refinement algorithm the option `cegar-abstraction-refinement-algorithm` must be set to `SIGMA` as follows:

```
:set cegar-abstraction-refinement-algorithm SIGMA
```

Abstract Transition Relation Refinement

Once the failure states have been "split" as described above the abstract transition relation has to be updated. The abstract transition relation is recomputed locally by distributing the incoming and outgoing transitions of the failure state on the new abstract states. Hence, we minimize the cost of abstract transition refinement. The pseudo code of the Distribute algorithm is given in Figure 6.3.

6.2.5 Certified Pairs Inference Methods

In order to infer certified pairs for our PCegar algorithm (see Section 4.4), we implemented the options presented below. Observe that one does not need to enable all of the following options in order to see the benefit of our PCegar algorithm.

Initial Certification

A simple method to infer certified pairs is called *initial certification*. The method relies on the obvious fact that we have $I \subseteq post^*(I)$ and $F \subseteq pre^*(F)$. Therefore, for a given set of abstract states $\widehat{Q}$ we test if an abstract state $\widehat{q} \in \widehat{Q}$ satisfies $\widehat{q} \subseteq I$. If this is the case, we add $\widehat{q}$ to $\widehat{X}_-$. Likewise, if we have $\widehat{q} \subseteq F$ we add $\widehat{q}$ to $\widehat{X}_+$.

To enable this certified pair inference method, the option `initial-certification` must be set as follows:

Distribute $(S, \widehat{q}, \widehat{q}', \widehat{q}'')$

Input: A transition system S, the failure state $\widehat{q}$, the new abstract states $\widehat{q}'$, and $\widehat{q}''$.

```
1  in = incoming_transitions(q̂)
2  out = outgoing_transitions(q̂)
3  for each transition →̂ in in do
4     ŝ = source(→̂)
5     if there exists s ∈ ŝ and q′ ∈ q̂′ such that s → q′ do
6        add a transition from ŝ to q̂′ in the abstraction
7     done
8     if there exists s ∈ ŝ and q″ ∈ q̂″ such that s → q″ do
9        add a transition from ŝ to q̂″ in the abstraction
10    done
11 done
12 for each transition →̂ in in do
13    t̂ = target(→̂)
14    if there exists t ∈ t̂ and q′ ∈ q̂′ such that q′ → t do
15       add a transition from q̂′ to t̂ in the abstraction
16    done
17    if there exists t ∈ t̂ and q″ ∈ q̂″ such that q″ → t do
18       add a transition from q̂″ to t̂ in the abstraction
19    done
20 done
```

Figure 6.3: The abstract transition relation refinement pseudo code.

```
:set initial-certification
```

Split Certification

Another method to infer certified pairs for our PCegar algorithm is called *split certification*. In order to extend certified pairs, this method relies on the current abstraction refinement heuristic. As discussed in Section 4.3.2 the direct refinement heuristic can be used to infer certified pairs. For instance a new abstract state can be added to $\widehat{X}_-$ when direct refinement is used together with the forward counterexample analysis algorithm. Likewise, a new abstract state can be added to $\widehat{X}_+$ when direct refinement is used together with the backward counterexample analysis algorithm.

To enable this certified pair inference method, the option `split-certification` must be set as follows:

```
:set split-certification
```

Feasibility Certification

During the analysis of an abstract counterexample, subsets of reachable concrete states (from the concrete initial states), and subsets of the coreachable concrete states (from the concrete final states) are computed. The forward (resp. backward) analysis algorithm can be modified to test the abstract states at each step. More precisely, after the execution of the instruction Line 5 in both forward, and backward counterexample analysis algorithms Figure 6.1, and Figure 6.2 respectively, we have $\widehat{q}_i = F[i]$ we can add $\widehat{q}_i$ to X_-, or X_+ respectively.

To enable this certified pair inference method, the option `feasibility-certification` must be set as follows:

```
:set feasibility - certification
```

Must Transitions & Closure

In Section 4.3.1 we have discussed the inference of certified pairs with the help of must transitions and the closure operation. This method has been implemented and can be enabled using the `cegar-use-must-and-clo` option. To enable this certified pair inference method the option must be set of follows:

```
:set cegar-use-must-and-clo
```

6.2.6 Running the CEGAR Loop

Once an abstraction of an AltaRica node have been computed, the CEGAR loop can be triggered to start the verification process. The CEGAR loop is started using the command `ar-cegar-verify`. The syntax of the command is:

```
:ar-cegar-verify <AltaRica Node> [count]
```

The arguments are:

Argument	Description
AltaRica Node	Name of the AltaRica node to analyze.
count	The maximal number of iteration to perform.

If the argument count is not specified, the loop will continue until a decision is made: either the error states are not reachable from the initial states, or they are reachable and an abstract trace is returned. If the argument count is specified, the loop will stop when it has performed "count" iterations (or a decision has been made before). If the cegar loop did not conclude the options (refinement heuristics, ...) can be modified and the `ar-cegar-verify` can be relaunched.

6.2.7 Other CEGAR Commands

For benchmarks and analysis purpose we implemented the following options:

- `cegar-print-iteration-stats`

- `cegar-abstraction-print`

The `cegar-print-iteration-stats` when enabled prints to the user statistics at each iteration of the CEGAR loop. The statistics are the following:

Abstraction states	Description
Trimming	Name of the AltaRica node to analyze.
CounterExample	The maximal number of iteration to perform.
State Refinement	The maximal number of iteration to perform.
Transition Refinement	The maximal number of iteration to perform.

The `cegar-abstraction-print` command output to a file (using the dot format) the abstraction obtained at each iteration of the CEGAR loop.

6.3 Benchmarks

Now that the have presented our implementation of the Cegar and PCegar algorithm, we present in more details the benchmarks proposed in Chapter 4.6, and also present another set of benchmarks we have performed on a satellite navigation system.

6.3.1 The Burns Model

We start by going over a benchmark model of the Chapter 4.6: the Burns model. This model is an AltaRica modelization of the Burns mutual exclusion algorithm. We have seen in Chapter 4 that on this example the PCegar algorithm outperformed the Cegar algorithm using the Post and Pre refinement methods.

Direct Forward Analysis

The direct forward analysis refinement method is our implementation of the `Post` refinement heuristic presented in Chapter 4.6. In Mec 5 this refinement heuristic is set using the following options:

```
:set cegar-abstraction-refinement-algorithm Direct
:set cegar-ce-analysis Forward
```

In this setting Mec 5 will execute the Cegar algorithm with the `Post` refinement heuristic. To enable the use of certified approximations and use the PCegar algorithm the following options have been set:

```
:set cegar-use-must-and-clo
:set split - certification
:set  feasibility - certification
:set initial-certification
```

Abstract State Space. In Figure 6.4 we have represented the evolution of the state space of the abstractions manipulated by the Cegar and PCegar algorithms. As expected the use of certified approximations (the PCegar algorithm) reduces the abstract state space

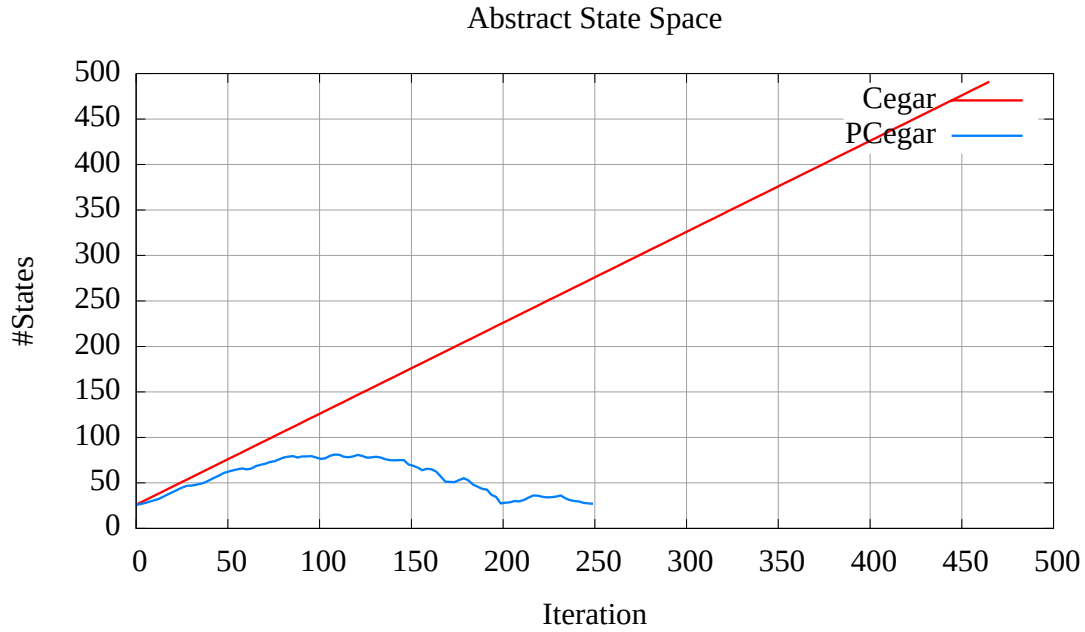


Figure 6.4: Abstraction state space evolution of the abstract Burns model with the `Post` refinement method .

required by the algorithm. Note that, we also applied the classical abstraction reduction to Cegar, which removes abstract states that are not reachable or not co-reachable. Yet the PCegar algorithm thanks to certified approximations sees its abstraction state space increase less rapidly. Moreover after 110 iterations, the PCegar algorithm reached its largest abstraction, and the following abstractions are continually smaller. On the other hand we observe that the Cegar algorithm does not prune abstract states until its last iteration when it can conclude.

Abstract CounterExample Length. A direct consequence (and advantage) of the reduced abstract state space is the length of the abstract counterexamples manipulated by our verification algorithms. This is illustrated in Figure 6.5 where the Y-axis stands for the number of abstract states in a counterexample, and the X-axis represents the successive iterations. An important observation is the length of the counterexamples produced by the Cegar algorithm: as we can see they increase similary to the abstract state space. We can observe a BMC (bounded model checking) behavior: counterexamples of a given length are analyzed before longer counterexamples. This comes from the BFS counterexample search algorithm. However, we observe that PCegar using the same setting manipulates counterexamples that are, and remain, smaller. Likewise, this is clearly an advantage the pruning enabled by certified approximations.

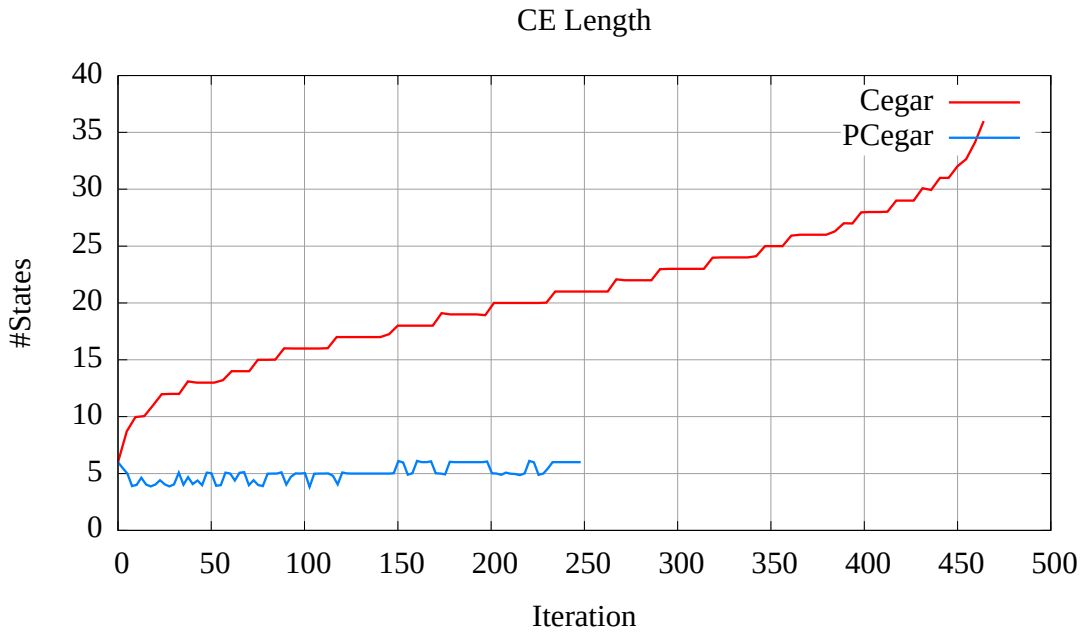


Figure 6.5: Abstract counter example length of the Burns model during verification with the `Post` refinement method.

Direct Backward Analysis

The direct backward analysis method is our implementation of the `Pre` refinement heuristic the dual of `Post` presented in Chapter 4.6. This refinement heuristic is set in Mec using the following options:

```
:set cegar-abstraction-refinement-algorithm Direct
:set cegar-ce-analysis Backward
```

To enable the use of certified approximations the options given in the previous section are set.

In Figure 6.6 we have represented the evolution of the abstraction state space during the execution of the Cegar and PCegar algorithms. Here we observe that in the first few iterations both algorithms generate refined abstractions of similar size, yet after 15 iterations the PCegar algorithm is able to prune away abstract states that Cegar cannot. Here we once again observe that Cegar can sometimes prune away abstract states but less frequently and in a smaller proportion that PCegar.

In Figure 6.7 we also represented the abstract counterexample length obtained by our algorithm during the verification process. We observe the same "stairway" behavior of the Cegar algorithm whereas the PCegar algorithm keeps analyzing counterexample of a stable length.

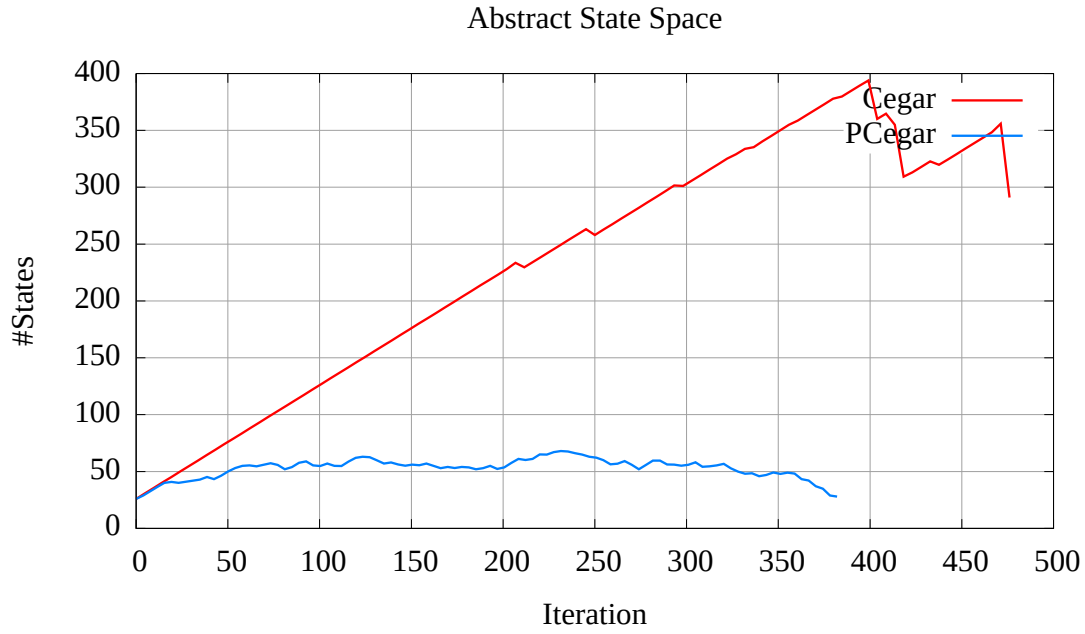


Figure 6.6: Abstraction state space evolution of the abstract Burns model with the `Pre` refinement method.

This example clearly exhibits the advantages of the PCegar algorithm discussed in Chapter 4.4:

- *Some useless refinements can be avoided.*
- *Counterexamples are shorter.*

In order to illustrate the behavior of PCegar when there is a feasible counterexample, we present a case study.

6.3.2 Satellite Formation Flying Case Study

During ANR SPACIFY project, an AltaRica modelization of a (simplified) satellite navigation system was developed and verified. This "satellite formation flying" model manages two satellites that can, as the name suggests, fly in formation. One of the satellites is the *master*, and the other one is the *slave* satellite. The master satellite communicates with the ground station, and relays navigation commands to the slave satellite. The navigation system of each satellite is composed of the flight control software along with hardware components such as: a star tracker, an inertial measurement unit, and a cold gaz propulsion system. There is an additional component that models the communication between the master and slave satellites. Each hardware component can be in various states from `Off` to `Failed`. All of these components are duplicated, and a failure injection component is introduced in the model in order to simulate physical or logical errors.

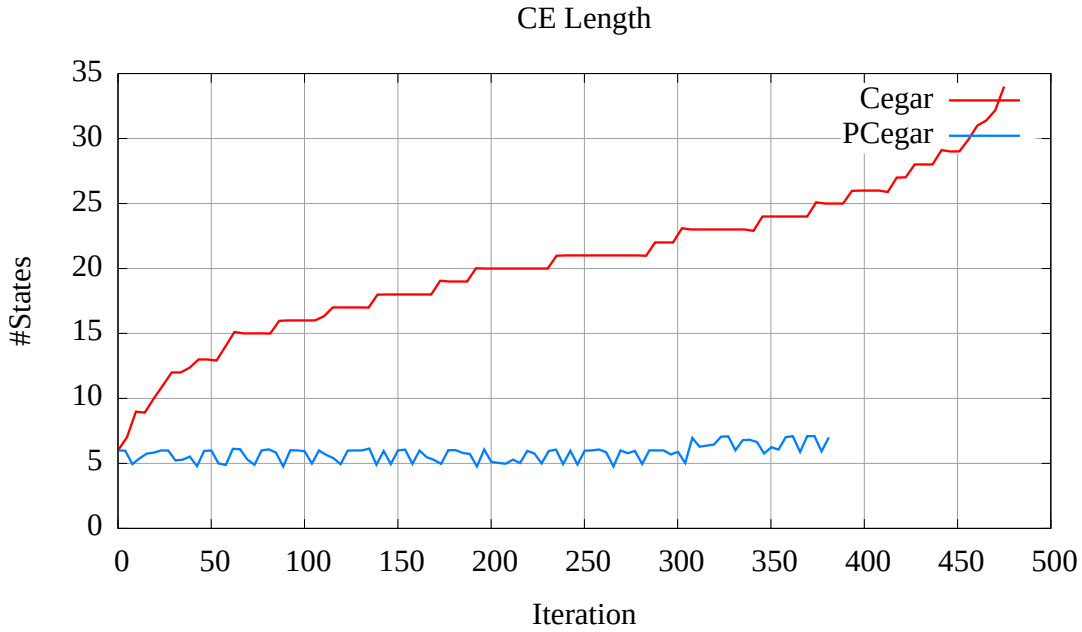


Figure 6.7: Abstract counter example length of the Burns model during verification with the `Pre` refinement method.

The flight control software switches navigation modes upon reception of commands from the ground station. It may also automatically switch to a degraded mode depending on the status of the hardware components. The complete modelization of this satellite formation flying system comprises 15 AltaRica nodes, for a total of 700 lines of code. The state space of the semantics of the model has 4.310^{22} states, and the reachable part (from the initial configurations) contains 8.110^{7} states.

The case study was limited to the verification of five safety properties on the satellite formation flying model. These safety properties express relationships between the navigation modes and the status of hardware components. Among these five safety properties, four are satisfied by the model, and one isn't. The property that is violated is the following: If the cold gaz propulsion system has completely failed (i.e., there is no redundancy available anymore for this component), then the navigation mode is `CollisionAvoidance`. This invariant ensures that the navigation system is in a safe mode when the gaz propulsion system is down.

We benchmarked the Cegar and PCegar algorithms on this invariant in order to determine their respective behavior when a counterexample existed in the model. Using the `Post` refinement heuristic both algorithms failed to find the bug, but using the `Pre` refinement heuristic PCegar was able to determine the existence of a counterexample in 26 iterations whereas the Cegar algorithm was inconclusive after 80 iterations[2].

[2] At this point the allocated memory was exhausted and Mec stopped.

In Figure 6.8, we have represented the abstract state space of the abstraction used by Cegar and PCegar. Surprisingly we note that no pruning was performed by either algorithm, and yet PCegar was conclusive. In Figure 6.9, we illustrate the evolution of the $\widehat{Q}_+$ states during the execution of PCegar. We observe that at each iteration a (new) single abstract state is added to this set. Yet, despite this continuous extension of the $\widehat{Q}_+$ set, no pruning is performed.

In Figure 6.10, we illustrate the evolution of the length of the abstract counterexamples obtained during the verification process performed by Cegar and PCegar. Here we can observe that PCegar keeps analyzing abstract counterexamples of length 3 whereas Cegar analyzes counterexamples of increasing length.

The counterexample in the model is of length 14, and PCegar could determine its existence using an abstract counterexample of length 3. This is due to two advantages of PCegar:

- *Counterexamples are shorter*: PCegar's counterexamples are factors of Cegar's counterexamples.
- *Counterexamples are more likely to be feasible*: Cegar may pick a counterexample that is spurious even though the corresponding counterexample of PCegar is feasible (or, worse, Cegar's spurious counterexample has been completely eliminated in PCegar).

Here, PCegar takes advantage of the factorization induced by certified pairs: for instance when we can reach an abstract state of $\widehat{Q}_+$ we know that there exists a path from this state to a "bad" state, and we can conclude, but Cegar has to find the entire path to conclude.

6.4 Concluding Remarks

In this chapter, we have presented our implementation of the Cegar and PCegar algorithms in Mec 5, and detailed some benchmarks on academic models as well as an industrial one. The benchmarks confirmed the expected gain, but more importantly showed that PCegar algorithm manipulates certified approximations that are much smaller than their standard counterparts. These results, validate our pruning methods, and our efforts to implement our CEGAR algorithms.

We should note that it is possible to treat our examples with other tools. For example, ARC [Poi00, Arc10] outperformed our CEGAR algorithms (particularly on the Spacify model). This comes from the efficient symbolic representation of states used in ARC: Decision Diagrams. Experiments showed that this structure is in many cases more concise than the BDDs used in Mec 5. The objective of our experiments was to compare methods and not implementations. We have seen that PCegar algorithm finds shorter counterexamples and needs less iterations than Cegar. So the comparison would be the same if we implemented the two algorithms in a more efficient model-checker.

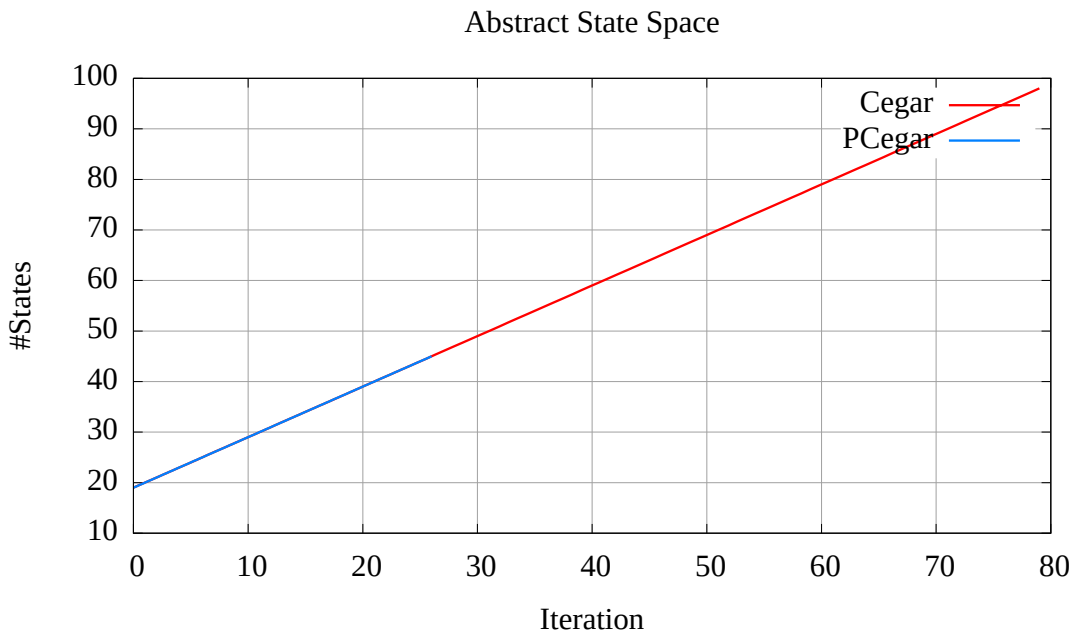


Figure 6.8: Abstraction state space evolution of the abstract Spacify model with the `Pre` refinement method.

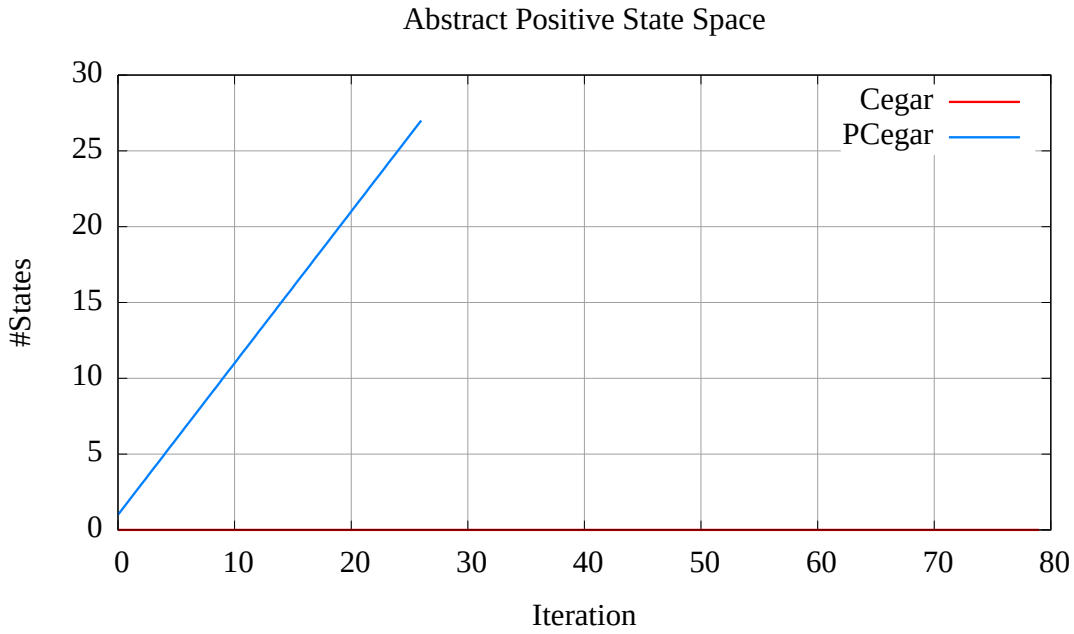


Figure 6.9: Positive certified states evolution of the Spacify model during verification with the `Pre` refinement method.

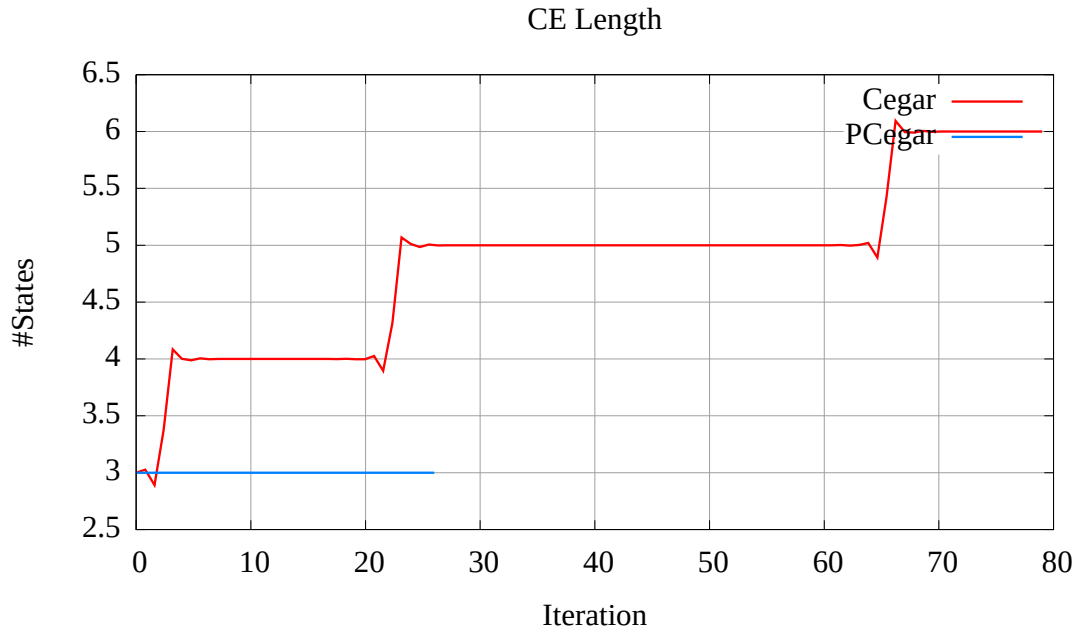


Figure 6.10: Abstract counter example length of the Spacify model during verification with the `Pre` refinement method.

Chapter 7

Conclusion

Formal verification appeared as an answer to the growing demand of safety in the design of critical systems. Among other methods, model checking became a popular and competitive method as it permits the *automatic* verification of systems. Over time, different approaches to model checking where proposed to tackle the state-space explosion problem. Explicit-state exploration methods, symbolic methods, and abstraction-based methods, each approach built upon it predecessor a new paradigm that improved the possibilities of model checkers. CEGAR is one of the most successful abstraction-based methods that permits model checkers to scale up and verify large pieces of software, and models.

This thesis presents improvements of the CEGAR method on its key structure: abstractions. The improvements proposed are of two kinds. First is an abstraction pruning method that eliminates abstract states during the execution of the CEGAR loop. The second is an abstraction method for hierarchical transition systems.

The PCegar algorithm presented in Chapter 4 is the first contribution of this thesis. It permits the use of an under-approximation of the system state space to determine the reachability of "bad" states of the system, without loosing soundness (Proposition 4.3). This is possible because, we take advantage of abstract states proved to hold only reachable (or coreachable) concrete states: certified pairs. Certified pairs allow a large pruning of abstractions. As a consequence the counterexample search algorithm focuses on counterexamples that are more likely to be feasible: kernel paths. The kernel paths are in fact factors of an abstract counterexample, more importantly, a kernel path can be a factor of many abstract counterexamples. Therefore, verifying a kernel path proves spurious one or more abstract counterexamples of a usual abstraction. Certified pairs also provide an easy way to determine reachability (the $\mathcal{W}$ condition page 58). This condition can be tested directly on the structure of our certified approximation efficiently. We also proposed methods for the inference of certified pairs (Chapter 4 Section 4.3) that profit from the computations done by a CEGAR algorithm and the abstraction analysis method. Fi-

nally, experiments validated our method as the PCegar algorithm outperformed the Cegar algorithm on a variety of academic models and an industrial one.

In Chapter 5 we have considered modular aspects of the AltaRica language that we have abstracted in the form of hierarchical transition systems. This model permits a modular representation of a system, that is now a standard way to implement industrial systems. First we have considered the case without priorities. We have given an abstraction method for this model using cover abstraction of each component. With this abstraction we have seen that we can verify an abstract counterexamples locally on each transition system using the VerifyHierarchicalPath algorithm. Equipped with our abstraction and this efficient abstract counterexample verification method, we could obtain a HierarchicalCegar algorithm. We then turned our attention to hierarchical transition systems with priorities. In this setting, even the abstraction step is complex since the use of cover abstractions may result in an unsound abstraction once priorities are applied (see Chapter 5 Section 5.3). To tackle this issue, we have introduced a concept of neat covers. This particular type of cover abstraction is sensitive to outgoing transitions of states: only states with the same set outgoing events can be abstracted by the same abstract state. With this kind of abstractions, we can use again the VerifyHierarchicalPath algorithm to verify a hierarchical transition system. Finally, we proposed two methods to obtain neat covers of AltaRica nodes.

Perspectives

The work we have presented here opens many perspective. We now list and discuss some of them.

Certified Pairs Inference

The efficiency of the PCegar algorithm increases as certified pairs are discovered. The methods we proposed are semantic-based. The advantage of this semantic approach is its genericity: it can be applied to any model whose semantic is represented by a transition system. On the other-hand, this inference method can be costly, and does not take advantage of the structure of the model. Syntactic refinement methods such as interpolation can take advantage of this model structure. Combining both approaches would greatly benefit our algorithm and widen its applicability to a larger set of model checkers. To this end, we need to find an interpolation method that allows us to infer reachability (or coreachability) properties from the predicates it generates.

Abstraction of Hierarchical Transition Systems

Our hierarchical abstraction method of a hierarchical transition system preserves its schema. Being able to modify the schema would give more flexibility to our HierarchicalCegar algorithm. For instance, it would be interesting to abstract a subtree of the hierarchy by a single node labeled with a suitable transition system. Deciding which

subtree should be abstracted is already a challenging task. But finding a suitable transition system is even more difficult for hierarchical transition systems with priorities.

Generalizing Abstraction of Hierarchical Transition Systems

In Chapter 5 a abstraction method have been defined for priority free hierarchical transitions systems, and another one for hierarchical transitions systems with priorities. The first relies on the use of cover abstractions (see Chapter 4 Section 4.1), and the second requires the use of neat covers (see Chapter 5 Section 5.3). We have looked at the parallel composition of non-hierarchical components, that is one of the simplest instances of a hierarchical system. In this case it is possible to work with a weaker condition than neat covers. In short, we have considered an equivalence relation where two states are equivalent, if and only if, their outgoing transitions are labeled with incomparable events (w.r.t. the priority relation). This abstraction method is not sound for arbitrary hierarchical transition systems. Yet, we believe that a weaker condition that captures both priority free setting and neat covers condition can be found.

Pruning of Hierarchical Transition Systems Abstractions

A natural extension of the work we presented here is a pruning extension of the abstraction of hierarchical transition systems. Priorities once again make this task difficult, but even without priorities some challenges remain. As pruning may eliminate abstract states, it becomes difficult to ensure soundness. Intuitively, one must be able to associate different feasible kernel paths to determine if they represent at least one common abstract counterexample.

Applicability of Hierarchical Transition Systems

A hierarchical transition system is a modular and concise representation of a system. An advantage of this model is its potential applicability to verification of industrial systems. A concrete example is a the interlocking device controlling train traffic in a station. Such a device manages the signals and switches of a railway track, in order to guarantee safe movements of trains. Recently, formal methods have been used to verify this type of equipment [BMM+08, Bou11]. The MatLab Simulink StateFlows and the Scade Suite, are (and have been) used to develop the software for this device. The hierarchical transition systems model considered in this thesis is suitable and easily adaptable to MatLab Simulink StateFlows. We believe that our hierarchical approach of CEGAR can be applied to such models. This is a challenging task, but very motivating as it can benefit both the industrial and academic communities.

Bibliography

[AGPR00] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The AltaRica Formalism for Describing Concurrent Systems. *Fundamenta Informaticae*, 40:109–124, 2000.

[AN82] A. Arnold and M. Nivat. Comportements de Processus. *Colloque AFCET Les Mathématiques de l'Informatique*, pages 35–68, 1982.

[And97] H.R. Andersen. An Introduction to Binary Decision Diagrams. *Lecture Notes, available online, IT University of Copenhagen*, 1997.

[Ang87] D. Angluin. Learning Regular Sets From Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[Arc10] Arc web page, 2010. http://altarica.labri.fr/forge/projects/arc.

[Auc08] N Aucouturier. Language Interpolation For Abstraction-Based Model-Checking. Master's thesis, Université Bordeaux 1, LaBRI, 2008.

[BCC+03] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:117–148, 2003.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDFW08] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing Abstractions. *Fundamenta Informaticae*, 89(4):369–392, 2008.

[Ber09] R. Bernard. *Analyses de Sûreté de Fonctionnement Multi-Systmes.* PhD thesis, Université Bordeaux 1, LaBRI, 2009.

[BHJM07] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast. *Int. Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[BHMR07] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *Proc. of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*, pages 300–309, New York, NY, USA, 2007. ACM.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[BKS07] T. Ball, O. Kupferman, and M. Sagiv. Leaping Loops in the Presence of Abstraction. In *Proc. of the 19th Int. Conf. Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 491–503. Springer, July 2007.

[BKY05] T. Ball, O. Kupferman, and G. Yorsh. Abstraction for Falsification. In *Proc. of the 17th Int. Conf. Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 67–81. Springer, July 2005.

[BMM+08] S. Behnia, A. Mammar, J. M. Mota, N. Breton, P. Caspi, and P. Raymond. Industrialising a proof-based verification approach of computerised interlocking systems. In *Proc. of the 11th Int. Conf. on Computer System Design and Operation in the Railway and Other Transit Systems (COMPRAIL'08)*, pages 143–152. WIT Press, 2008.

[Bou11] J.L. Boulanger. *Techniques Industrielles de Modélisation Formelle pour le Transport.* Hermes Science Publications, 2011.

[BPG08] M.G. Bobaru, C.S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proc. of the 20th Int. Conf. on Computer Aided Verification (CAV'08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2008.

[BPR03] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Int. Journal on Software Tools for Technology Transfer (STTT'03)*, 5(1):49–58, 2003.

[BR01] T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. of the 13th Int. Conf. Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[Bry86] R.E. Bryant. Graph-Based Algorithms For Boolean Function Manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of the 14th Int Conf. on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[CFC+09] Y.-F. Chen, A. Farzan, E.M. Clarke, Y.-K. Tsay, and Wang B?-Y. Learning Minimal Separating DFA's for Compositional Verification. In *Proc. of the 15th Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2009.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. CounterExample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[CGP03] J. Cobleigh, D. Giannakopoulou, and C. Păsăreanu. Learning Assumptions for Compositional Verification. In *Proc. of the 9th Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[Cla08] Edmund M. Clarke. The Birth of Model Checking. *25 Years of Model Checking*, pages 1–26, 2008.

[CMCHG96] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic Model Checking. In *Proc. of the 8th Int. Conf. Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer, 1996.

[COYC03] S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke. Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach. *Electronic Notes in Theoretical Computer Science*, 89(3):417–432, 2003.

[Cra57] W. Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[CS07] S. Chaki and O. Strichman. Optimized L*-Based Assume-Guarantee Reasoning. In *Proc. of the 13th Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2007.

[Das] Safety designer (BPA/DAS). http://www.schwindt.eu/angebot/bpa/safety-designer.html.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[GC06] A. Gurfinkel and M. Chechik. Why Waste a Perfectly Good Abstraction? *Proc. of the 12th Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06)*, 3920:212–226, 2006.

[GGP07] M. Gheorghiu, D. Giannakopoulou, and C. Păsăreanu. Refining interface alphabets for compositional verification. In *Proc. of the 13th Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2007.

[GJ02] P. Godefroid and R. Jagadeesan. Automatic Abstraction Using Generalized Model Checking. In *Proc. of the 14th Int Conf. on Computer-Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2002.

[GMF07] A. Gupta, K. L. McMillan, and Z. Fu. Automated Assumption Generation for Compositional Verification. In *Proc. of the 20th Int. Conf. on Computer Aided Verification (CAV'07)*, volume 5505 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2007.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs of infinite systems with PVS. In *Proc. of the 9th Int. Conf. Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, July 1997.

[GV04] A. Griffault and A. Vincent. The Mec 5 Model-Checker. In *Proc. of the 9th Int. Conf. on Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 248–251. Springer, 2004.

[GWC06] A. Gurfinkel, O. Wei, and M. Chechik. Yasm: A Software Model-Checker for Verification and Refutation. In *Proc. of the 18th Int. Conf. Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 170–174. Springer, 2006.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. of the 29th Symposium on Principles of Programming Languages (POPL'02)*, pages 58–70. ACM, 2002.

[Hol97] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[JM05] R. Jhala and R. Majumdar. Path Slicing. In *Proc. Conf. Programming Language Design and Implementation (PLDI'05)*, volume 40, pages 38–47, New York, NY, USA, June 2005. ACM.

[Lar89] K.G. Larsen. Modal Specifications. In *Proc. Int. Workshop Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246, June 1989.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):1–44, 1995.

[LSC95] K.G. Larsen, B. Steffen, and Weise C. A Constraint Oriented Proof Methodology Based on Modal Transition Systems. In *Proc. of the 1st Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 17–40. Springer, May 1995.

[LT88] K.G. Larsen and B. Thomsen. A Modal Process Logic. In *Proc. of the 3rd Annual Symposium on Logic in Computer Science (LICS'88)*, pages 203–210. IEEE Computer Society, May 1988.

[LX90] K.G. Larsen and L. Xinxin. Equation Solving Using Modal Transition Systems. In *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 108–117. IEEE Computer Society, June 1990.

[McM04] Kenneth L. McMillan. An Interpolating Theorem Prover. In *Proc. of the 10th Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 16–30, 2004.

[McM06] K. McMillan. Lazy Abstraction with Interpolants. In *Proc. of the 18th Int. Conf. Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[Mec10] Mec 5.4 web page, 2010. http://altarica.labri.fr/forge/projects/mec.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. of the 2nd Int. Joint Conference on Artificial Intelligence*, pages 481–489. Stanford University Stanford, CA, USA, 1971.

[Pag04] C. Pagetti. *Extension Temps Réel d'AltaRica.* PhD thesis, Ecole Centrale de Nantes, 2004.

[PLA10] AltaRica Public Models Web Page, 2010. http://altarica.labri.fr/forge/projects/public-models.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer-Verlag, New York, NY, USA, 1985.

[Poi00] G. Point. *AltaRica: Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement.* PhD thesis, Université Bordeaux 1, LaBRI, 2000.

[PU59] MC Paul and SH Unger. Minimizing the Number of States in Incompletely Specified Sequential Switching Functions. *IEEE transactions on computers*, pages 356–367, 1959.

[RCB] D. Richard, K. R. Chandramouli, and R. W. Butler. Cost effective use of formal methods in verification and validation. Int. Workshop on Verification and Validation, 2002.

[RS89] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proc. of the 21st annual ACM symposium on Theory of computing(STOC'89)*, pages 411–420, New York, NY, USA, 1989. ACM.

[SG04] S. Shoham and O. Grumberg. Monotonic Abstraction-Refinement for CTL. In *Proc. of the 10th Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 546–560. Springer, April 2004.

[Sim] Simfia v2. http://www.apsys-software.com/simfia.xhtml.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties using Induction and a SAT-solver. In *Proc. of the 3rd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2000.

[Vin03] A. Vincent. *Conception et réalisation d'un vérificateur de modèles AltaRica.* PhD thesis, Université Bordeaux 1, LaBRI, 2003.

[WLBF09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.