THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Victor ROUSSANALY

Efficient verification of real time systems

Verification efficace de systèmes en temps réel

Thèse présentée et soutenue à Rennes, France, le 30 novembre 2020
Unité de recherche : IRISA , Team SUMO
Thèse N° :

Rapporteurs avant soutenance :

Thao Dang	Directrice de Recherche - CNRS
Étienne André	Professeur - LORIA

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Prénom Nom	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Frédéric Herbreteau	Maître de Conférence
	Olivier H. Roux	Professeur - LS2N/École Centrale de Nantes
	Thao Dang	Directrice de Recherche - CNRS
	Étienne André	Professeur - LORIA
	B Srivathsan	Associate Professor - Chennai Mathematical Institute
Dir. de thèse :	Nicolas Markey	Directeur de Recherche - CNRS
Co-dir. de thèse :	Ocan Sankur	Chargé de recherche - CNRS

Invité(s) :

Prénom Nom	Fonction et établissement d'exercice

Acknowledgement

Il y a plus de 3 ans, Nicolas et Ocan m'ont pris en stage quand j'étais en recherche de sujet. Pendant ces 3 années, ils m'ont guidé avec bienveillance et patience, tout en me montrant l'étendue de la recherche et de leurs domaines. Pour cela je les remercie infiniment.
Je dois aussi remercier Nathalie pour m'avoir aiguillé vers les méthodes formelles et le model checking, en 2015, alors que j'étais un peu perdu. C'est grâce à elle, ainsi que Luc et David, que j'ai eu mes premiers contacts dans ce domaine.
Je remercie les rapporteurs, pour avoir pris le temps de relire mon manuscrit, ainsi que le reste du jury pour accepter d'assister à cette soutenance. Je remercie aussi mes parents, qui m'ont soutenu toutes ces années. Mon père qui m'a conseillé sur la voie à suivre quand j'avais 16 et que j'ai annoncé que je voulais être chercheur, ainsi que ma mère qui m'a donné la liberté de faire ce que je voulais. Comme j'ai pu vous le répéter, tout commence avec un "professeur 123" à l'arrière d'une voiture. C'est là la fondations de mon éducation, et c'est vous qui m'avez poussé à ne pas m'arrêter à la facilité et à me donner des défis. Je remercie aussi mon grand frère qui a essayé de s'intéresser à ce que je faisais.
Bien évidemment, je tiens à remercier l'équipe SUMO pour son accueil. Tout d'abord les permanents, qui m'ont fourni conseils et remarques constructives : Blaise, Eric B., Eric F., Hervé, Loïc, Nathalie et Thierry. Je tiens aussi à remercier les non-permanents, post-docs et thésards ensemble. En premier, ceux qui étaient présent à mon arrivée et qui ont déjà fini leur thèse : Engel, Karim, Hugo, Matthieu, Sihem et The Ahn. Puis, tous ceux que j'ai vu arriver : Abdul, Adrian, Anirban, Arthur, Bastien, Emily, Leo, Rituraj et Suman.
Je remercie aussi Gilles et Sophie pour qui j'ai pu donner des cours. Ces heures m'auront permis d'améliorer ma pédagogie, de prendre du recul, et de faire des pauses durant des phases frustrantes de cette thèse. Aussi mon CSID, Patricia et François, pour avoir pris le temps de m'écouter.
Je remercie aussi les autres doctorants avec qui j'ai pu partager et échanger. D'abord ceux de me promo sur Rennes : Alexandre, Antoine, Joris, Thibaut et Timothée. Puis aussi aux autres doctorants avec qui j'ai pu m'entretenir : Clémence, Eva, Justine, Jules et Manu.
Enfin je remercie le reste de mes amis, et ne pouvant faire une liste exhaustive, je vais

noter ceux qui m'ont soutenu durant l'écriture de ce manuscrit : Aurélien, Nicolas L., Gurvan, Chloé, Maïlys, Charlotte, Fanny, Marco, Romain, Rémy, Burd et Loum.

Résumé

Alors que les processus issus de l'informatique sont de plus en plus présents dans notre vie, notre dépendance sur ces systèmes automatisés est devenue de plus en plus importante. Ces processus automatisés nous entourent et nous guident dans notre vie de tous les jours, que ce soit à travers nos ordinateurs, smartphones, nos voitures, ou même des objets connectés, ou même des machines simples mais de plus en plus automatisées telles qu'une machine à café ou un réfrigérateur connecté. Bien sûr ils sont aussi présents de manière plus discrète, que ce soit en dirigeant nos institutions financières, en guidant le trafic routier, en choisissant le rythme d'un pacemaker, ou tout simplement en gérant le trafic de nos abondantes communications. Les exemples sont multiples et abondants, et leur nombres ne fait que croitre. En effet, cette année encore, nous avons pu constater à quel point nos besoins dans ces outils numériques est grandissant, avec une demande grandissante pour le télétravail et les cours en ligne, créant ainsi un besoin pour des plateformes et des protocoles. De plus nous rentrons aussi dans l'ère de la conduite automatisée, avec des voitures de plus en plus autonomes qui se démocratisent.

Bien sûr avec ces systèmes de plus en plus critiques qui reposent sur l'automatisation, cela crée aussi un besoin de confiance dans ces systèmes. Cela s'est illustré par plusieurs fiascos au cours du siècle passé. On peut citer l'exemple de la sonde Mars Climate Orbiter qui fut perdue avant d'arriver en orbite autour de Mars. Cette perte de 328 millions de dollars a été déterminée comme étant due à une erreur d'unité de mesure, le système de navigation étant conçu pour utiliser le système de mesure impérial, plutôt que le système métrique. De manière similaire, un des exemples les plus connus est celui du vol 501 de la fusée Ariane 5, durant lequel la fusée explosa. La cause de cette défaillance d'un coût estimé à 500 millions de dollars est le stockage d'un entier 64 bit dans un espace de stockage de 16 bits. On peut aussi citer des satellites russes qui en 1983 confondent une réflexion du soleil avec des missiles américains, manquant ainsi de peu de déclencher une nouvelle guerre mondiale. D'autres exemples à plus petite échelle sont tous aussi critiques, comme une faille de sécurité dans des pacemakers, permettant à une personne extérieure d'en prendre le contrôle [49]. Il a aussi été montré que des changements de quelques pixels pouvaient tromper des logiciels de reconnaissances visuels, de manière à ce qu'un feu rouge

ne soit plus reconnu par exemple, ce qui peut causer des problèmes pour des véhicules autonomes [48].

Pour protéger les vies humaines, et pour réduire le cout en cas de défaillance, ces problèmes peuvent être traités en amont, durant la phase de développement. Plusieurs méthodes ont été développées et peuvent être appliquées durant la phase de conception.

Assistants de preuves. Ce sont des outils puissants pour résoudre ce problème de vérification. Ici, le comportement attendu du système est décrit comme un théorème mathématique. Ensuite, ces assistants de preuve, tels que Coq, Isabelle ou encore Why3, peuvent être utilisés pour prouver que l'implémentation vérifie ce théorème, et donc que le comportement est bien celui attendu. Ici, la difficulté réside déjà dans le fait que l'on doit exprimer le comportement comme une propriété mathématique. Ensuite l'assistant de preuve a généralement besoin d'être guidé pour finir la preuve. Cela veut dire que l'on va généralement faire appel à un expert qui maitrise cet assistant de preuve pour vérifier cette propriété. En retour, cela permet d'exprimer énormèment de propriétés.

Solveurs. Ces outils sont utilisés quand la propriété peut être exprimée comme une formule logique simple. Par exemple, les solveurs SMT sont utilisés quand le problème peut être exprimé par une formule logique du premier ordre. Avoir cette expression en formule logique peut être ardu, mais les solveurs SMT ont l'avantage de ne pas avoir besoin d'être guidés et sont automatisés. En réalité, les assistants de preuve utilisent souvent des solveurs SMT, tels que CVC4 ou OpenSMT pour résoudre leurs problèmes.

Tests. On peut aussi effectuer des simulations de notre système. On crée ainsi un modèle représentant notre système et on effectue des tests sur cette simulation, c'est-à-dire on simule des exécution du système pour un environnement donné. Cela peut aussi être fait sur le système fini plutôt que sur une simulation, mais cela n'est pas toujours possible, et si le système a un problème sur certains tests, le développer à nouveau peut avoir un coût trop élevé. Donc cela requiert d'avoir une simulation assez précise du système. Un des soucis de cette approche est que l'ensemble des tests effectués doit être suffisamment exhaustif et représentatif pour avoir une confiance élevée dans le système. L'avantage de cette approche est qu'elle requiert peu de connaissance sur le système si un simulateur est fourni.

Model checking. Dans cette thèse, nous nous focalisons sur cette approche. Cela consiste à construire un modèle représentant notre système et à spécifier notre problème dans une logique adaptée, par exemple LTL ou CTL comme dans [11]. Ici, la difficulté vient du fait qu'il faut réussir à construire un modèle qui capture l'expressivité de notre modèle tout en gardant une taille raisonnable, de manière à ce que les algorithmes de model checking puissent être appliqués en un temps acceptable. Par exemple, NuSMV est un model checker qui utilise plusieurs autres outils et solveurs pour construire et vérifier des propriétés CTL sur des modèles dont la taille est raisonnable. Bien sûr, chaque outil prend différents types de modèles en entrée. On peut avoir des modèles simples, comme des systèmes de transitions dans lesquels les états du systèmes sont explicites, qu'on peut ensuite enrichir avec de la composition asynchrone, des canaux de communication, des variables, des contraintes sur ces variables ou encore de la synchronisation. Toute cette sémantique peut ensuite être convertie en un système de transition sur lequel des algorithmes de model checking peuvent être appliqués. Le problème ici est que ces algorithmes sont dépendants du nombre d'états dans le système de transition et que celui-ci croit lorsque l'on rajoute cette sémantique. Par exemple, la composition de plusieurs systèmes ou l'ajout de variables fait croitre la taille du système de transition de manière exponentielle. Ce problème de l'explosion combinatoire de l'espace d'états à vérifier est bien connu et plusieurs approches existent pour y circonvenir. Premièrement, éviter le parallélisme et limiter le nombre de variables dans la conception du modèle permet de faciliter la résolution. Ensuite des approches utilisant des Diagrammes de Décision Binaires (ou BDDs) sont souvent utilisées pour ordonner les variable booléennes à évaluer.

Abstraction et raffinement

Une idée proposée dans [21] pour limiter cette explosion est de limiter le nombre d'états, même si cela implique de perdre de la précision sur notre modèle, et d'avoir un modèle qui ne correspond pas exactement au système. En effet, une abstraction de notre modèle concret est une représentation de notre modèle dans laquelle plusieurs états sont groupés en un seul état. Dans certains cas, l'abstraction est bien choisie et le modèle abstrait vérifie exactement les même propriétés que le modèle concret. Mais bien souvent, l'abstraction est trop grossière et des chemins qui n'étaient pas présents dans le modèle concret apparaissent. Par exemple on peut voir sur la fig. 1 un modèle de feu tricolore ainsi que son abstraction. Mais l'abstraction pour ce feu fait apparaître un chemin qui ne

passe jamais dans *go* qui n'existe pourtant pas dans le modèle concret original.

La solution proposée pour ce problème est une boucle de raffinement. En effet, si l'abstraction est trop grossière et si un chemin apparait dans l'abstraction qui réfute la propriété recherchée, alors les algorithmes de model checking retournent un contre-exemple. Ce contre-exemple peut être comparé avec le modèle concret pour vérifier si cela est en effet un contre-exemple, ou un artéfact issu de l'abstraction. Si il n'est présent que dans l'abstraction, on peut l'utilise pour raffiner l'abstraction de manière à ce que ce contre-exemple n'y apparaisse plus, et on peut relancer les algorithmes de model checking sur ce nouveau modèle abstrait en étant convaincu que ce contre-exemple ne sera pas réutilisé. Cette boucle qui alterne entre model checking sur abstraction et raffinement d'abstraction est appelée CEGAR pour "Counter-Example Guided Abstraction Refinement".

Ce procédé est de plus en plus utilisé pour répondre à la demande de model checking sur des modèles de plus en plus complexes, contenant de plus en plus d'états. Par exemple dans [27] dans un algorithme pour vérifier des abstractions de programmes C. Ou encore dans [22] où une approche CEGAR est présentée pour vérifier des propriétés quantitatives.

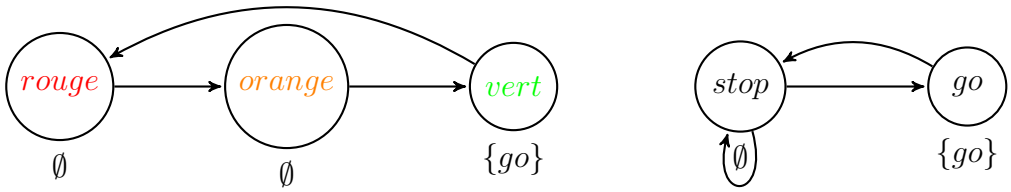


FIGURE 1 – Un exemple d'un modèle de feu tricolore, ainsi que son abstraction dans laquelle les états *rouge* et *orange* ont été groupés en un état stop.

Systèmes temporisés

Dans la plupart des automatisations, le comportement du système doit répondre à des contraintes de temps, telles que des temps de réponses ou des deadlines. On les appelle alors systèmes en temps réel ou systèmes temporisés.

Cela est important en model checking car créer un système temporisé demande non seulement un développement correct du logiciel, mais requiert aussi de se conformer à des contraintes de temps. La complexité croissante de ces systèmes fait que leur vérification est d'autant plus importante et compliquée. Cela a pu être vu sur le système de freinage de la Toyota Prius, où une erreur logicielle a engendré un délai dans la réponse du frein. On peut

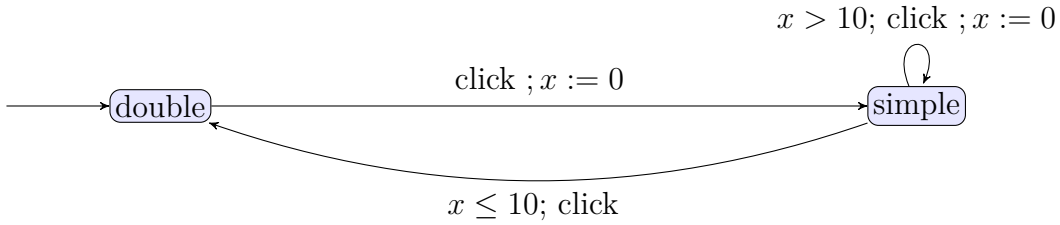


FIGURE 2 – Exemple d'un automate temporisé pour modéliser un double clic.

aussi noter l'attaque Spectre où des différences dans le délai de réponse des programmes permet d'inférer des données privées.

Plusieurs modèles peuvent être utilisés pour modéliser ces systèmes, tels que les réseaux de Petri temporisés [43] ou alors les systèmes hybrides. Le modèle auquel nous nous intéressons ici est celui des automates temporisés. Ceux-ci peuvent être vus tels que des extensions des automates finis auxquels on rajoute des horloges, c'est-à-dire des variables réelles dont la valeur augment de manière synchrone lorsque l'on attend dans un état. Plusieurs outils ont déjà été développés pour ces automates, tel que Uppaal [35], Imitator [8], TChecker ou Kronos [50]. Ces automates peuvent être utilisés pour prouver que le comportement d'un système temporisé est bien celui attendu, comme dans [7] où une extension des ces automates est présentée pour répondre à des attaques telles que Spectre, mais ils peuvent aussi être utilisés pour générer un contrôleur par exemple [19]. La fig. 2 montre un exemple d'un tel automate, modélisant une souris d'ordinateur sur laquelle on peut double-cliquer. L'horloge x sert ici à mesurer le temps entre deux clics consécutifs, nous faisant passer dans l'état double si les deux actions se sont suivies assez rapidement.

Dans cette thèse, nous nous concentrons sur le problème d'accessibilité d'un état, ou problème de sûreté. Il a été montré dans [4] que ce problème est PSPACE-complet, en utilisant un découpage de l'espace des valeurs d'horloge en régions, c'est-à-dire des ensembles d'horloges atomiques qui vérifient les même propriétés. Une amélioration existe, utilisant un découpage de cet espace des valeurs en parties plus larges appelées zones.

Dans cette thèse, nous développons deux algorithmes pour répondre au problème d'accessibilité dans les automates temporisés, en utilisant la boucle CEGAR. L'un d'entre eux se base sur les méthodes énumératives traditionnellement utilisées par Uppaal et TChecker, tandis que l'autre se base sur des méthodes symboliques et des réductions de formules booléennes. Nous avons implémenté ces algorithmes, puis nous les avons évalués sur des benchmarks en les comparant à Uppaal, qui est un des outils les plus utilisés pour l'accessibilité sur des automates temporisés. De plus nous présentons aussi un algorithme

basé sur de la recherche heuristique pour répondre à un problème de synthèse de contrôleur dans des cas particuliers de planification de mouvement.

Table of Contents

Introduction **13**

1 State of the art **21**

1.1 Preliminaries . . . 21

1.1.1 Transitions systems . . . 21

1.1.2 Clocks, valuations and constraints . . . 21

1.1.3 Timed automata and semantics . . . 22

1.1.4 Semantics of timed automata . . . 24

1.2 Regions and zones . . . 25

1.2.1 Reachability problem and examples . . . 26

1.2.2 Regions . . . 28

1.2.3 Zones . . . 30

1.2.4 Difference Bound Matrices . . . 33

1.2.5 Extrapolation . . . 35

1.3 Counterexample Guided Abstraction Refinement . . . 39

2 Enumerative Algorithm **43**

2.1 Abstraction refinement . . . 43

2.1.1 Quantization abstraction . . . 43

2.1.2 Interpolants . . . 44

2.2 Enumerative algorithm . . . 48

2.2.1 Core idea of the CEGAR loop . . . 48

2.2.2 Abstract forward reachability: AbsExplore . . . 49

2.2.3 Refinement: Refine . . . 53

2.2.4 Implementation discussion . . . 56

3 Symbolic Algorithm **59**

3.1 Notation . . . 59

3.1.1 Clock predicate abstraction. . . . 60

3.1.2 Reduction . . . 61

3.2 Successor Computation 65
3.2.1 Intersection 65
3.2.2 Time Successors 65
3.2.3 Reset 67
3.3 Model-checking algorithm 69
3.3.1 Abstract successors computation 69
3.3.2 Spurious trace detection 70
3.3.3 Refinement Procedure. 71

4 Experiments **75**
4.1 Benchmarks 76
4.1.1 CSMA/CD 76
4.1.2 Monoprocess Scheduling Analysis 77
4.1.3 Multiprocess Stateful Scheduling Analysis 78
4.1.4 Asynchronous Computation 79
4.2 Overall results. 81

5 Funnel automata: a heuristic approach **87**
5.1 Funnel automata 87
5.1.1 Control funnel 87
5.1.2 Motion planning with funnels 90
5.1.3 Funnel automata 92
5.1.4 Pick and place problem 94
5.2 Algorithm for a pick-and-place problems 96
5.2.1 A heuristic approach 96
5.2.2 Results 98

Conclusion **101**

Bibliography **105**

Introduction

Verification

As digital processes became part of our everyday lives, our reliance on automatized processes grew accordingly. They guide us most of our time, through our computers and smartphones, but also through our connected devices, our cars, or some of our everyday appliances such as coffee machines or connected fridge. Of course they also drive our financial institutions, are routing our everyday commute through traffic lights, are choosing the appropriate rhythm for a pacemaker, and overall are responsible for routing and handling most of our communications. And that is not taking into account the growing popularity of automated driving. The examples of our dependency on this automation are almost infinite and this list is always growing. Just recently we have seen an urge for classes and overall most services to go online, creating a need for platforms, protocols and tools.

Of course this ever growing reliance comes with a consequential need for trust in these systems. And it was shown many times that blind trust was not enough, as these processes did not always work as intended. In the worst cases, the cost can be counted in human lives and millions/billions of euros. Some of these examples are infamous, such as the Mars Climate Orbit Crash, where a navigation system was designed to use imperial units instead of the metric system. One of the most infamous example might be the Ariane V failed launch, where a 64 bit number was stored in a 16 bit space, or maybe the Soviet early warning satellite in 1983, that picked up sunlight reflections off cloud-tops and mistakenly interpreted them as missile launches in the United States, almost causing the start of World War III. These examples are extreme and might seem like unique occurrences, but some are not that extreme and can occur in our daily lives. For example it has been shown that changing only a few pixels in an image can change the output in a recognition software, such that a traffic light might not be recognizable, creating a risk of importance for automatized driving [48]. Another example is a lack of security on a medical pacemaker that can be accessed remotely [49]. While the most famous examples concern critical systems, there are some seemingly benign sub optimal issues that, while not a danger,

may cost a few euros at each time, multiplied by thousand or millions of usages.

In order to protect human lives and to reduce the costs, and to deal with this problem in the early phase of development of these systems, several formal methods have been developed to make sure that these automated processes behave as planned.

Proof assistants are a powerful tool to solve this problem. In this method, the expected behavior of the system is given as a mathematical theorem. Then, using proof assistants such as Coq, Isabelle or Why3, one can prove that the implementation of the system respects this theorem. Here the difficulty is twofold: we have to first express the expected behavior as a mathematical property, and then we have to guide the proof assistant to prove this property. This means that this method, while allowing us to check all kind of properties, requires an expert to guide the proof.

Solvers are used when the property can be expressed as a simpler formula. Satisfiability Modulo Theories (SMT) solvers are used when the problem can be described as a formula in first order logic. While expressing the property as a first order logic formula can be hard, SMT solvers do not need human supervision. Actually most proof assistants use a SMT solver to facilitate the proof. The most known SMT solvers are CVC4 and OpenSMT.

Tests and simulations can also be done. This consists at looking at the execution of the system for a given environment, and a given initial state. This can be simulated on a model representing the system, or sometimes this can be done on the system itself. Note that this is not always a possibility and that if a critical design error is made apparent on the real system, the cost of redesigning it might be too much. So we usually rely on accurate simulation of our system. Of course, depending on the property we want to prove, we can generate different sets of test cases. The main problem of this approach is that the tests cannot be exhaustive, and this cannot be considered as an absolute proof that the process will behave as expected. On the other hand, this can be done with little to no information on the system, considering it as a black box. The difficulty here is to generate test cases that will give us a high confidence that the system satisfies a property.

Model checking [20] is a method used when we can build a model of our system, and when we can specify our property in a given logic, for example LTL or CTL as shown in [11]. In this thesis, we will focus mainly on this approach to verification. Here the difficulty stems from the fact that we want a model that captures all the expressiveness of our system

while having a reasonable size, so algorithms can still work on it in a reasonable time. For example, NuSMV is a model checker that uses several other tools and solvers to create models whose size are manageable. Those models can be finite transition systems where all the states of the system are explicit, or they can be enriched with more expressiveness, such as adding bounded variables, with operations and constraints on these variables to have a program graph, or having multiple systems that communicates with each other, either through handshaking or through communication channels. In any cases, all this added semantic can be transformed into a finite transition systems, where checking a property is easier. Of course, each model checker takes different kind of model as an input. So choosing an appropriate model, with the right semantic to model our system is key, and can determine the model checker we use. Of course, this model can be more or less precise in the way it represents the system, and as such be more or less expressive. This is relevant since the algorithms used depend on the size of the model. So a precise model tends to have many states represented, meaning that the model checking algorithms used will take a longer time to conclude. Moreover, this number of states tends to grow exponentially whenever the number of variables increases. This well known problem is usually referred as the *state space explosion*. This is usually seen as we have multiple systems that are interacting with each other, creating a state space that is the product of the state space of each individual system. Many approaches are used to avoid this problem. First, in the design phase, avoiding parallelism or limiting the number of variables will reduce the impact of this problem. Then in the model checking part, using symmetry or solving the problem on smaller independent instances of our system in order to reduce the size of the model we check [23]. For example, using Binary Decision Diagrams (BDDs) to order the variables and the order on which we treat them is a common method to solve this problem.

Counter-example Guided Abstraction Refinement

Another idea to answer this problem was proposed in [21]. The idea is to reduce the number of states even further, by merging them together. This process, called an *abstraction*, reduces the number of states to explore, but consequentially it may create paths that do not exist in the original model. In [21], the authors consider only properties that pertain to all the paths of the models, like safety properties. Since the abstracted model contains all the paths from the original model, if this property is verified on the abstracted model, then this property also holds for the original concrete model. On the

other hand, if this property is not verified on the abstract model, since it is a property on all the paths, then there is counter-example that shows that there exists a path that disproves this property. If this path is realizable in the original model, then we know that the property was false, otherwise a new abstraction can be built using the existing abstraction and this counter-example path. This abstraction still over approximates the possible paths, but every path in this new abstraction is a path of the previous abstraction, meaning that no new path was added. Moreover this abstraction is defined such that it does not contain the counter-example paths. This means that this process can be repeated until an answer is found, giving us a finer abstraction at each iteration.

This process has been used more and more in model checking to answer the growing sizes of the models. For example, in [27], an algorithm designed to have automatic verification abstraction of C programs using the CEGAR paradigm is presented. In [22] a CEGAR approach is introduced to verify quantitative properties. These algorithms have been implemented into tools and provided good results, for example BLAST [28] or SLAM [12]. In all these cases, a CEGAR approach was introduced in order to have algorithms that can finish faster.

Real-time systems and Timed automata

In most automated systems, the behavior depends on time in a quantitative manner. We consider *real-time systems*, that is systems that are subject to real-time constraints such as response times and operational deadlines. This is of particular interest in model checking as designing such systems not only involves correct development of software, but also requires to answer timing constraints. The growing complexity of these systems means that checking their correctness has become more and more relevant and arduous. In term of critical systems, this can be seen in the the Toyota Prius brake system software error that created a delay of the braking system. This error could cost lives, but some other errors could also cost money. For example most manufacturing chains and assembly lines are now automated using a Programmable Logic Controller (PLC). Indeed, most of these assembly lines now use safety PLC in order to detect when the emergency stop is necessary. This need for considering time is even more visible when looking at the recent Spectre attack, where time is used as a side channel to attack a system and obtain private information.

These systems are even harder to design, as the classical testing and debugging of

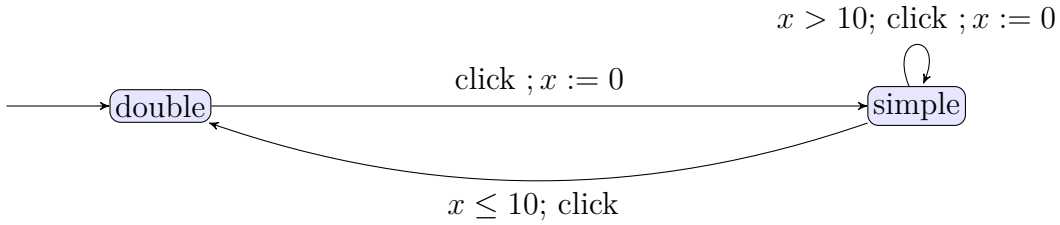


Figure 3 – Example of a timed automaton for a double click.

real-time systems is often difficult, as reproducing a timing error in a given environment might be hard. Moreover time cannot be stopped while debugging and analyzing the system. Let us also remember that nowadays, most systems are distributed systems, which is already a difficulty for designing concurrent software systems that is inherited when designing distributed real-time systems. Most importantly, execution time and timing constraints are not usually taken into account in software development. For all these reasons, a lot of errors in real-time systems happen in the design phase.

In order to detect these errors using model checking, many models can be used such as Timed Petri nets [43], or linear hybrid systems [2], but one of the most common models are *timed automata*[3][5], and they are the focus of this thesis. They can be seen as extensions of finite automata to which we added synchronized clocks, that can be compared and reset. Several model checking tools are available for timed automata, such as Uppaal [35], Imitator [8], TChecker or Kronos [50]. These automata can be used to both prove the correctness of timed protocols, but also to generate a scheduling or a controller for a given timed system [19]. There are also extensions of these timed automata to allow for more expressiveness and to solve harder problems, such the weighted timed automata [14] or parametric timed automata [7], but in this thesis we will only consider timed automata. In Figure 3, we show an example of a timed automaton modeling a the behavior of the button of a computer mouse, with one clock x. If a click happens, then we go to the state "simple" and start measuring time by resetting the value of x to 0. If another click happen in the next 10 time units, measured by x, then we consider it to be a double click and we go to the state "double". Otherwise a click happens after 10 time units and as such it is just another simple click and we reset the value of x to check if this new click can become a double click.

In this work, we will only consider one kind of property and that is the property of safety, which means that we will only check whether or not a state can be reached. The problem of reachability for timed automata has been shown to be PSPACE-complete in

[4]. The method commonly used to solve it will consider the state we are in as well as a set of clocks valuation. The first approach consists at partitioning the space of possible clock valuations into sets of clocks that cannot be distinguished by any integer clock constraints. These atomic sets are called *regions*, and we can use them to build a transition system. Another approach consists at tracking all the possible clock valuations we can be in after each transition. These sets are defined as convex areas in the value space of clocks and are called *zones* [18]. The algorithm used to solve reachability depend on this number of zones. And this number is exponential in the number clocks since it is the dimension of this value space. Moreover, it also depends on how the partitioning is done, meaning that this requires to limit the state space that we explore.

Abstraction for timed automata

Model checkers like Uppaal already implement abstraction. These abstractions are necessary to build ourselves a finite system on which we can apply our algorithms. Moreover, these abstractions are also sound and complete, meaning that if a state is reachable in the abstracted model, it is reachable in the concrete model, and vice versa. Moreover, abstractions that are not sound have already been developed for timed automata. This means that some states might be seen as reachable in this abstracted model while this is not the case in the original concrete model. This is done using the CEGAR paradigm described above. In [40], an abstraction that tries to remove all clocks before reintroducing them by using refinements as they are needed. Another approach in [37] consists at removing the different constraints on clocks in the timed automata, and then refining it by identifying the necessary constraint, and then splitting the states between those who satisfies this necessary constraint and those who do not. Although so far these methods have not be proven to be more efficient than the ones used in well established model checker such as Uppaal, this thesis aim to use these methods to have algorithms that are more efficient for reachability in timed automata.

Contribution of this thesis

In this thesis, we present two algorithms that apply the CEGAR approach to timed automata. The first chapter is a description of the well known algorithms implemented in Uppaal, as well as several other CEGAR approach to timed automata that have been tried

before. In the second chapter we show an algorithm that we developed and implemented that applies the CEGAR approach to reachability timed automata. This algorithm is based on an enumerative approach to the problem, similarly to the way Uppaal solves reachability. In the third chapter we explain another algorithm that we developed that is solving the same problem using a symbolic approach. This means that instead of having an enumerative exploration, we instead reduce the problem to a formula, abstracted with the CEGAR approach. We then solve the reachability problem using the CEGAR loop as well as BDD algorithms. The results for both of these algorithms are shown in Chapter 4, where we also detail several benchmarks that we developed to test these algorithms. Finally in Chapter 5 we remind a way to construct a timed automaton that can be used to generate a controller for a dynamical system, and we show a heuristic approach that we implemented to solve this problem more efficiently.

Chapter 1

State of the art

1.1 Preliminaries

1.1.1 Transitions systems

A good way to model a system is to define a state space where the state contains all the information necessary for the exploration, and then to define transitions between all these states, representing the different changes that can occur within a system. Formally, this simple model is the *transition system*, as defined in [11]. It is a set of states that are connected by edges that are labeled with a letter.

Definition 1.1.1. *A transition system is a tuple* $(S, s_0, \Sigma, \rightarrow)$ *where:*

— S *is a set of states*

— $s_0 \in S$ *is the initial state*

— Σ *is the alphabet of actions*

— $\rightarrow \subseteq S \times \Sigma \times S$ *is the set of transitions.*

This can also be called a *finite automaton*. In model checking, the transition system is used to express the semantics of models that are complex. Indeed, the semantic of a transition system is self apparent, as we can go from one state s to the next state s' with the action α if and only if there is a transition $(s, \alpha, s') \in \rightarrow$, also noted $s \xrightarrow{\alpha} s'$. Then a state s is reachable from s_0 if there is a sequence of states $s_1, s_2, ..., s_n$ and actions $\alpha_0, \alpha_1, ..., \alpha_n$ such that there are transitions $s_i \xrightarrow{\alpha_i} s_{i+1}$ for all $i < n$ and $s_n \xrightarrow{\alpha_n} s$.

1.1.2 Clocks, valuations and constraints

In order to model timed systems, a common method is to use *clocks*, that is real-valued variables that evolve synchronously. Let us consider a finite set $\mathcal{C} = \{x_1, ..., x_N\}$ of *clocks*. We call a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$ a *clock valuation*. For a clock valuation v, a subset

$R \subseteq \mathcal{C}$, and a non-negative real d, we denote with $v[R \leftarrow d]$ the valuation w such that $w(x) = v(x)$ for $x \in \mathcal{C} \setminus R$ and $w(x) = d$ for $x \in R$, and with $v + d$ the valuation w' such that $w'(x) = v(x) + d$ for all $x \in \mathcal{C}$. This delay is used to represent an elapsed period of time, during which the values of the clocks grow synchronously.

For a set of valuations $S \subseteq \mathbb{R}^{\mathcal{C}}_{\geq 0}$, we extend those operations such that $S[R \leftarrow d] = \{v[R \leftarrow d] \mid v \in S\}$ and $S + d = \{v + d \mid v \in S\}$. We write $\vec{0}$ for the valuation that assigns 0 to every clock. Given $R \subseteq \mathcal{C}$, we define the *reset* of a valuation v, denoted by $v[R \leftarrow 0]$, as it intuitively follows: $v[R \leftarrow 0](x) = 0$ if $x \in R$, and $v[R \leftarrow 0](x) = v(x)$ otherwise. An *atomic constraint* is a formula of the form $x \bowtie k$ or $x - y \bowtie k$ with $x, y \in \mathcal{C}$, $k \in \mathbb{N}$, and $\bowtie \in \{<, \leq, >, \geq\}$. We say that v satisfies the atomic constraint $x \bowtie k$, denoted by $v \models x \bowtie k$ if and only if $v(x) \bowtie k$. And v satisfies the atomic constraint $x - y \bowtie k$, denoted by $v \models x - y \bowtie k$ if and only if $v(x) - v(y) \bowtie k$.
A *guard* or *constraint* g is a conjunction of atomic constraints. For a positive integer n let us write $g = \bigwedge_{0 \leq i \leq n} g_i$ a constraint with g_i an atomic constraint for $0 \leq i \leq n$. A valuation v satisfies g, denoted $v \models g$, if and only if for all $0 \leq i \leq n$, $v \models g_i$. In other words all atomic constraints hold true when each $x \in \mathcal{C}$ is replaced with $v(x)$. Let $[\![g]\!] = \{v \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid v \models g\}$ denote the set of valuations satisfying g. We write $\Phi_{\mathcal{C}}$ for the set of guards built on $\mathcal{C}$.

1.1.3 Timed automata and semantics

Timed automata (TA) were introduced by Alur and Dill in [6] and [4], as an extension of finite automata as a convenient formalism to model real-timed systems. In this thesis, timed automata are our main formalism used to model systems. In essence, a timed automaton is a finite automaton to which one adds clocks that are modeled by real-valued variables, and the behavior of the automaton is restricted by constraints on these clocks. Those clocks can be reset or compared with integer constants. Nowadays, the model that is studied a lot, as in [16], and especially here, is *timed safety automata.* These automata are used in the model checker Uppaal [35], and since they are the only ones we are studying here, we refer to them simply as timed automata.

Definition 1.1.2 (Timed automata)**.** *A* timed automaton $\mathcal{A}$ *is a tuple* $(\Sigma, \mathcal{L}, \mathsf{Inv}, \ell_0, \mathcal{C}, E)$ *where*

1. Σ *is a finite set of actions*
2. $\mathcal{L}$ *is a finite set of locations*
3. $\mathsf{Inv}\colon \mathcal{L} \to \Phi_{\mathcal{C}}$ *defines location invariants*

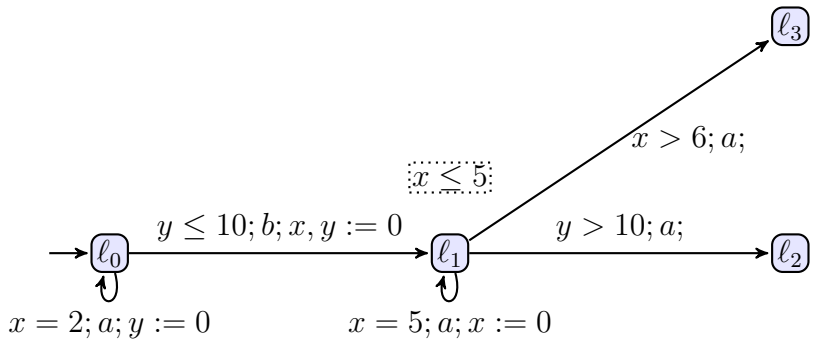


Figure 1.1 – An example of a timed automaton on $\{x; y\}$.

4. *$\mathcal{C}$ is a finite set of clocks*
5. *$E \subseteq \mathcal{L} \times \Phi_{\mathcal{C}} \times \Sigma \times 2^{\mathcal{C}} \times \mathcal{L}$ is a set of edges*
6. *$\ell_0 \in \mathcal{L}$ is the initial location.*

An edge $e = (\ell, g, a, R, \ell')$ is also written as $\ell \xrightarrow{g,a,R} \ell'$. For any location ℓ, we let $E(\ell)$ denote the set of edges $\{(\ell_1, g, a, R, \ell_2) \in E \mid \ell_1 = \ell\}$, that is the outgoing edges from ℓ.

Timed automata can be easily represented, as in Figure 1.1, where we represent a timed automaton with $\mathcal{C} = \{x, y\}$, $\Sigma = \{a, b\}$, $\mathcal{L} = \{\ell_0, \ell_1, \ell_2; \ell_3\}$, with resets being represented as an assignment to 0.

Although this is a definition of timed automata, this definition is unpractical to build large models, as the number of locations can be large. To simplify this construction, we use a synchronized product of timed automata.

Definition 1.1.3 (synchronized product of TAs)**.** *Let $N \in \mathbb{N}^*$. Given a set of TAs $\mathcal{A}_i = (\Sigma_i, \mathcal{L}_i, \mathsf{Inv}_i, \ell_{0,i}, \mathcal{C}_i, E_i)$, $1 \leq i \leq N$, and a set of actions Σ_s, the* synchronized product *of $\mathcal{A}_i$, $1 \leq i \leq N$, denoted by $\mathcal{A}_1 \parallel_{\Sigma_s} \mathcal{A}_2 \parallel_{\Sigma_s} \cdots \parallel_{\Sigma_s} \mathcal{A}_N$, is the tuple $(\Sigma, \mathcal{L}, \mathsf{Inv}, \ell_0, \mathcal{C}, E)$, where:*

1. *$\Sigma = \bigcup_{i=1}^{N} \Sigma_i$,*
2. *$\mathcal{L} = \prod_{i=1}^{N} \mathcal{L}_i$,*
3. *$\mathsf{Inv}((\ell_1, \ldots, \ell_N)) = \bigwedge_{i=1}^{N} \mathsf{Inv}_i(\ell_i)$ for all $(\ell_1, \ldots, \ell_N) \in \mathcal{L}$,*
4. *$\ell_0 = (\ell_{0,1}, \ldots, \ell_{0,N})$,*
5. *$\mathcal{C} = \bigcup_{1 \leq i \leq N} \mathcal{C}_i$,*

and E is defined as follows. For all $a \in \Sigma$, let σ_a be the subset of indices $i \in \{1, \ldots, N\}$ such that $a \in \Sigma_i$. For all $a \in \Sigma$, for all $(\ell_1, \ldots, \ell_N) \in \mathcal{L}$, for all $(\ell'_1, \ldots, \ell'_N) \in \mathcal{L}$, $\big((\ell_1, \ldots, \ell_N), g, a, R, (\ell'_1, \ldots, \ell'_N)\big) \in E$ if:

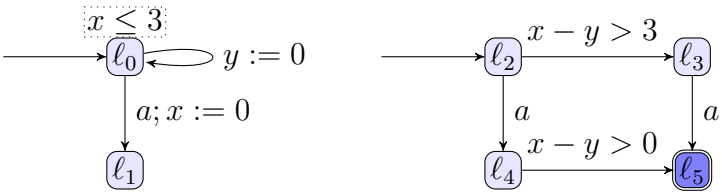


Figure 1.2 – Two TAs, denoted by $\mathcal{A}_1$ (on the left) and $\mathcal{A}_2$ (on the right).

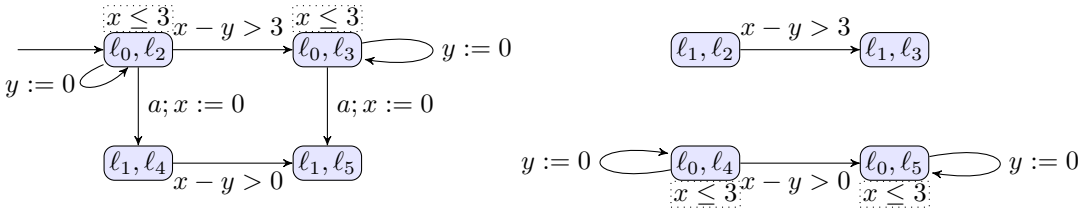


Figure 1.3 – The synchronized product $\mathcal{A}_1 \parallel_{\{a\}} \mathcal{A}_2$. All locations are represented.

— *if* $a \in \Sigma_s$*, then*

1. *for all* $i \in \sigma_a$*, there exist* g_i, R_i *such that* $(\ell_i, g_i, a, R_i, \ell'_i) \in E_i$*,* $g = \bigwedge_{i \in \sigma_a} g_i$*,* $R = \bigcup_{i \in \sigma_a} R_i$*, and,*
2. *for all* $i \notin \sigma_a$*,* $\ell'_i = \ell_i$*.*

— *otherwise (if* $a \notin \Sigma_s$*), then there exists* $i \in \sigma_a$ *such that*

1. *there exist* g_i, R_i *such that* $(\ell_i, g_i, a, R_i, \ell'_i) \in E_i$*,* $g = g_i$*,* $R = R_i$*, and,*
2. *for all* $j \neq i$*,* $\ell'_j = \ell_j$*.*

Of course, the number of locations in a product of timed automata is the product of the number of locations in each of its component, which can be a problem as most algorithms on TAs are dependent on the number of locations. Since the principal use of actions is to synchronize the products of TAs, we allow ourselves to not represent actions that will not be used in a product. An example of a product of TAs can be seen in Figure 1.3, where we represent the synchronized product of the two TAs shown in Figure 1.2.

1.1.4 Semantics of timed automata

As we explained Section 1.1.1, we will use transition systems to explain the semantics of timed automata. Since we are interested in timed models, we will consider *Timed Transition Systems* (TTS). In a timed transition system, the alphabet of actions is in $\Sigma \cup \mathbb{R}_{\geq 0}$, with the transitions (s, d, s'), $d \in \mathbb{R}_{\geq 0}$ called *delay transitions*. We will define the semantics of a timed automaton using a timed transition system.

Definition 1.1.4 (Semantics of a TA)**.** *Given a TA* $\mathcal{A} = (\Sigma, \mathcal{L}, \mathsf{Inv}, \ell_0, \mathcal{C}, E)$ *the semantics of* $\mathcal{A}$ *is given by the timed transition system* $(S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, \rightarrow)$*, with*

— $S = \{(\ell, v) \in \mathcal{L} \times \mathbb{R}_{\geq 0}^{\mathcal{C}} \mid v \models \mathsf{Inv}(\ell)\}$,

— $s_0 = (\ell_0, \vec{0})$,

— $\rightarrow$ *consists of the discrete transitions and continuous delay transition relations:*

1. *discrete transitions:* $(\ell, v) \xrightarrow{e} (\ell', v')$*, if* $(\ell, v), (\ell', v') \in S$*, and there exists* $e = (\ell, g, a, R, \ell') \in E$*, such that* $v' = v[R \leftarrow 0]$*, and* $v \models g$*).*
2. *delay transitions:* $(\ell, v) \xrightarrow{d} (\ell, v + d)$*, with* $d \in \mathbb{R}_{\geq 0}$*, if* $\forall d' \in [0, d], v + d' \models \mathsf{Inv}(\ell)$*.*

In other words, in a timed automaton, we can either wait in a location, as long as it satisfies its invariant, or we can take an edge if we satisfy its guard and apply its reset. We denote by $\mathcal{T}(\mathcal{A})$ the timed transition system associated with $\mathcal{A}$. Moreover we write $(\ell, v) \xrightarrow{e,d} (\ell', v')$ for a combination of a delay and discrete transition if $\exists v'' : (\ell, v) \xrightarrow{d} (\ell, v'') \xrightarrow{e} (\ell', v')$.

A *configuration* or *state* of a TA $\mathcal{A}$ is an element $q = (\ell, v) \in S$. A *run* of $\mathcal{A}$ is an alternating sequence of concrete configurations of $\mathcal{A}$ and pairs of edges and delays starting from the initial state s_0, of the form $q_0, (e_0, d_0), q_1, \cdots (e_n, d_n), q_{n+1}$ with $i = 0, 1, \ldots n$, $e_i \in E$, $d_i \in \mathbb{R}_{\geq 0}$ and $q_i \xrightarrow{e_i, d_i} q_{i+1}$. A *path* is a sequence of edges with matching endpoint locations.

A common definition for the synchronized product of two transition systems $T_1 = (S_1, s_{0,1}, \Sigma_1, \rightarrow_1)$ and $T_2 = (S_2, s_{0,2}, \Sigma_2, \rightarrow_2)$ over a set of actions Σ', also called rendezvous or handshake, can be found in [7]. It is defined as $T_1 \parallel_{\Sigma'} T_2 = (S_1 \times S_2, (s_{0,1}, s_{0,2}), \Sigma_1 \times \Sigma_2, \rightarrow)$ with $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ if and only if we have either:

1. $a \in \Sigma_s$, $s_1 \xrightarrow{a}_1 s_1'$ and $s_2 \xrightarrow{a}_2 s_2'$
2. $a \in \Sigma_1 \setminus \Sigma_s$, $s_1 \xrightarrow{a}_1 s_1'$ and $s_2 = s_2'$
3. $a \in \Sigma_2 \setminus \Sigma_s$, $s_1 = s_1'$ and $s_2 \xrightarrow{a}_2 s_2'$

Note that we then have $\mathcal{T}(\mathcal{A}_1 \parallel_{\Sigma'} \mathcal{A}_2) = \mathcal{T}(\mathcal{A}_1) \parallel_{\Sigma' \cup \mathbb{R}_{\geq 0}} \mathcal{T}(\mathcal{A}_2)$. We can also extend timed automata with bounded integers and Boolean variables, adding simple operations and comparisons. This will of course increase the size of the state space, since the states will now represent the location, the clocks valuation and the variables valuation.

1.2 Regions and zones

As we have seen, timed automata induce a timed transition system, which has an infinite uncountable number of states. We will consider the problem of reachability, and in

order to solve it, we will find an equivalent finite state space to explore, by finding a way to make our set of valuation discrete and finite.

1.2.1 Reachability problem and examples

Here, we will consider the problem of reachability in timed automata, that is whether a location ℓ or a set of locations T is reachable in a given timed automaton. This problem was proven to be decidable and in PSPACE [4]. It allows us to check for safety properties, but also for invariant properties and bounded liveness properties. These algorithms are implemented in the model-checker Uppaal [35] [17], first released in 1995. It implements most of the methods that we describe in this chapter. This problem is usually used in verification when the accepting set contains locations that models "error" states of the system. In this case, the property we are trying to verify is a safety property "an error state is never reachable". If the target location is reachable, then there is an *accepting run*, that is a run reaching the targeted state. If we can produce such a run, that is in fact a counter-example for our safety property. For example, in the model shown in Figure 1.1, the location ℓ_3 is not reachable, since $x < 5$ as long as we are in ℓ_1 and as such we can never take the transition to ℓ_3. On the other hand, the location ℓ_2 is reachable by looping at least twice in ℓ_1

Fischer protocol is used for mutual exclusion between multiple processes. That is, for n processes, numbered from 1 to n as P_i, make sure that no more than one of them can be in a critical section simultaneously. This can be easily represented with a synchronized product of timed automata $\mathcal{A}_i$ represented in Figure 1.4 [45]. Note that there are n clocks and a shared variable id, and that there is no synchronized action here. In fact the automata are synchronized using the shared variable id. The idea behind this model is that processes can reach a state `ready` only when id is set to 0. After 10 units of time or less, the process that reached ready has to go into `wait` so no process can go to ready anymore since the value of id is no longer 0. Once it reached `wait`, a process can either go back to `idle` or go into its critical section. In order to go to `idle`, the value of id needs to be changed to something else since the process reached `wait`, meaning that another process reached `wait` afterward. And in order to go into its critical section, it has to wait at least 10 units of time at least, which means that no process is in `ready` anymore. So the value id is set to the last process that reached `wait` and this process can only go into its critical section, and is the only one that can do so. Afterwards, it resets the value id to 0, every process is

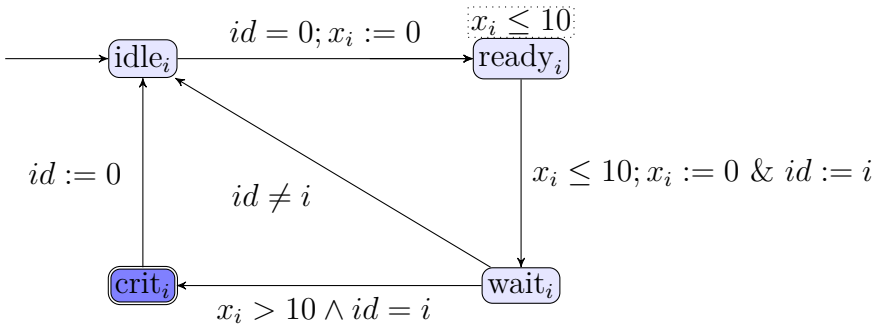


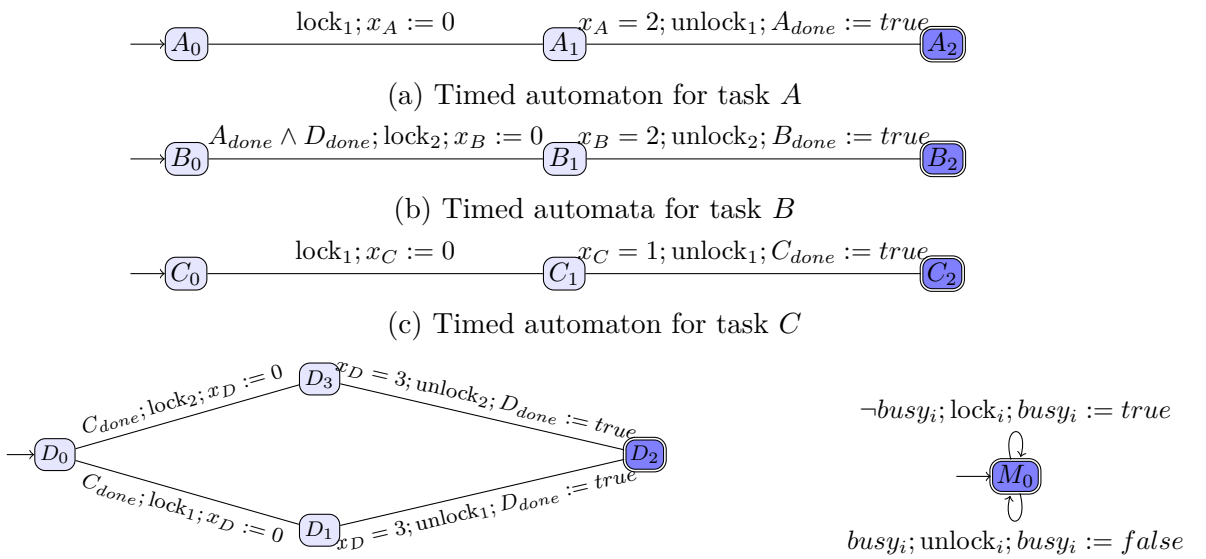
Figure 1.4 – A timed automaton $\mathcal{A}_i$ representing process P_i

in `idle` or are in `wait` and can only go to `idle` and this loop starts over. With this, we are fairly confident that we cannot reach a state where two processes are in their critical section at the same time, but we could use a model checker to prove this property.

Job Shop Scheduling problems can also be modeled with timed automata. Given a set of machines, a set of tasks and a recipe for each task, meaning constraints on the machine the task can be executed on, the order of execution for the tasks, and a duration for each tasks, this problems consists in finding a scheduling for these tasks. This problem can be solved using timed automata [1]. For example, let us assume 2 machines M_1 and M_2 and a set of tasks $\{A, B, C, D, E\}$ with the following constraints:

1. A takes 2 time units, and must be executed on M_1
2. B takes 2 time units, and must be executed after A and D, on M_2.
3. C takes 1 time units, and must be executed be executed on M_1.
4. D and E both take 3 time units, and can be executed on any machine.

Each task can be modeled by a timed automaton as shown in Figure 1.5. In this example, each task is modeled by a timed automaton, which first takes a machine among the available ones, then waits the duration of the task before releasing the machine and marking the task as complete. Each machine is also modeled by a one-state automaton that ensures that two different tasks cannot use the same machine at the same time. A model checker will tell us whether or not we can reach a target state, that is a state where all tasks are completed, but it will also give us a path to that state. This means that not only can we know that a scheduling is possible, but we also find such a scheduling. By adding a global clock that is never reset and by ensuring that the value of this clock is always smaller than a constant d, we can also check whether or not there is a scheduling that ensures that all



(a) Timed automaton for task A

(b) Timed automata for task B

(c) Timed automaton for task C

(d) Timed automaton for task D, identical for E

(e) Timed automaton for machine M_i

Figure 1.5 – Timed automata used to model the job shop scheduling problem. Each task and each machine is modeled by a timed automaton, our model being the synchronized products on the represented actions `lock` and `unlock`

the tasks are completed in less than d units of time, and find this scheduling if it exists.

1.2.2 Regions

In order for us to define a region we need a *clock ceiling function*, denoted by M, that associates to each clock x a maximum value $M(x)$.

Let us define $\sim_{reg}$ as an equivalence relation on clock valuations, which are functions of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$ defined by $u \sim_{reg} v$ iff:

— for all clocks x, $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$

— for all clocks $x, frac(u(x)) = 0$ if and only if $frac(v(x)) = 0$.

— for all clocks x and $y, frac(u(x)) \leq frac(u(y))$ if and only if $frac(v(x)) \leq frac(v(y))$.

Using this relation, we can write $[u]_{\sim_{reg}}$ the equivalence class of $u \in \mathbb{R}_{\geq 0}^{\mathcal{C}}$, and we can build a transition system $\mathcal{T}_{reg}(\mathcal{A}) = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, \Rightarrow)$ with $S = \mathcal{L} \times \mathbb{R}_{\geq 0}^{\mathcal{C}}$ and the transition $\Rightarrow$ defined as :

— $(\ell, [u]_{\sim_{reg}}) \overset{d}{\Rightarrow} (\ell, [v]_{\sim_{reg}})$ if $(\ell, u) \overset{d}{\rightarrow} (\ell, v)$ for d a non-negative number.

— $(\ell, [u]_{\sim_{reg}}) \overset{a}{\Rightarrow} (\ell, [v]_{\sim_{reg}})$ if $(\ell, u) \overset{a}{\rightarrow} (\ell, v)$ for $e \in E$.

From [16] we can find the proof for this theorem:

Theorem 1. *For a timed automaton $\mathcal{A}$ with an initial state ℓ_0, u_0, for $\ell_f \in \mathcal{L}$, and u_f a clock valuation, $(\ell_0, u_0) \to \cdots \to (\ell_f, u_f)$ in $\mathcal{T}(\mathcal{A})$ iff $(\ell_0, [u_0]_{\sim_{reg}}) \to \cdots \to (\ell_f, [u_f]_{\sim})_{reg}$ in this region transition system.*

This is simply due to the fact that for a valuation u and a valuation $v \in [u]_{\sim_{reg}}$, we have for any constraint g, $u \models g$ if and only if $v \models g$, and for any delay $d \in \mathbb{R}_{\geq 0}$, there exists a delay $d' \in \mathbb{R}_{\geq 0}$ such that $v + d' \in [u + d]_{\sim_{reg}}$. Now we have a transition system that has an infinite enumerable number of states. In order to limit ourselves to a finite transition system, we can use a *clock ceiling function*, denoted by M, that associates to each clock x a maximum value $M(x)$. We can then define $\sim_{reg,M}$ as an equivalence relation on clock valuations defined by $u \sim_{reg,M} v$ iff:

— for all clocks x, $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or both $u(x) > M(x)$ and $v(x) > M(x)$.

— for all clocks x, if $u(x) \leq M(x)$ then $frac(u(x)) = 0$ if and only if $frac(v(x)) = 0$.

— for all clocks x and y, if $u(x) \leq M(x)$ and $u(y) \leq M(y)$ then $frac(u(x)) \leq frac(u(y))$ if and only if $frac(v(x)) \leq frac(v(y))$.

By using this ceiling functions, we can also obtain regions, and in Figure 1.6 we show a representation of these regions. Every elementary surface with an open border, but also every border between this surfaces is a region. This means that, in this case with $M(x) = 3$ and $M(y) = 2$, we have 60 regions for only 2 clocks.

In this figure, we marked three valuations u, u' and u''. u is equivalent to v if and only if $\lfloor v(x) \rfloor = 2$, $\lfloor v(y) \rfloor = 1$, $frac(v(x)) = 0$ and $frac(v(y)) = 0$. This gives us that $[u]_{\sim_{reg}}$ is the single valuation u.
u' is equivalent to v iff $\lfloor v(x) \rfloor = 1$, $v(y) > 2$ and $frac(v(x)) \neq 0$. So $[u']_{\sim}$ is the set of valuations that satisfy $x = 1$ and $y > 2$.
u'' is equivalent to v iff $\lfloor v(x) \rfloor = 1$, $\lfloor v(y) \rfloor = 1$, $frac(v(x)) \neq 0$, $frac(v(y)) \neq 0$, $frac(v(x)) < frac(v(y))$.
This gives us that $[u'']_{\sim}$ is the open surface containing u'' on the figure, defined by the constraints $x > 1$, $x < 2$, $y > 1$, $y < 2$ and $x < y$.

Since the number of these regions is finite, we can similarly build a transition system $\mathcal{T}_{reg,M}(\mathcal{A})$ that has a finite amount of states. We can also note that this is an *abstraction* or in other words an over approximation of the transition system $\mathcal{T}_{reg}(\mathcal{A})$ that is itself an abstraction of $\mathcal{T}(\mathcal{A})$, we defined earlier. We also know that for a given TA $\mathcal{A}$, there exists a ceiling function M such that Theorem 2 still holds. For example the function M

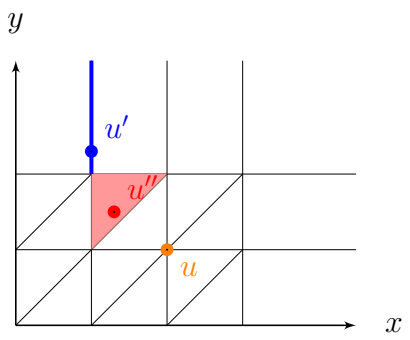


Figure 1.6 – A representation of regions

that associates x to the highest constant it is compared to would ensure the validity of this theorem. This transition system can also be referred to as a *region graph*. This means that the reachability problem for a timed automaton is equivalent to the reachability problem for the region abstraction. Let us denote by $\mathcal{R}(\mathcal{A}, M)$ the number of regions for the timed automaton $\mathcal{A}$ with a ceiling function M. In [4] it is explained that $\mathcal{R}(\mathcal{A}, M)$ can be bounded by $|\mathcal{C}|! \cdot 2^{|\mathcal{C}|} \cdot \prod_{x \in \mathcal{C}} (2M(x) + 2)$. Then the transition system is computed in time $O((|\mathcal{L}| + |E|) \cdot \mathcal{R}(\mathcal{A}, M))$ and the reachability problem can be solved in time $O((|\mathcal{L}| + |E|) \cdot \mathcal{R}(\mathcal{A}, M))$. Algorithm 1 details the reachability algorithm using regions. Given a timed automaton $\mathcal{A}$, an initial state (ℓ_0, u_0) and a target state (ℓ_f, r_f) with $\ell_f \in \mathcal{L}$ and r_f a region, this algorithm returns whether some state (ℓ_f, u_f) with $u_f \in r_f$ is reachable. We can also note that the reachability problem for timed automata is PSPACE-complete [4]. As we said, the number of regions is exponential which is a problem for the time complexity of the algorithm. To obtain an efficient algorithm, one aggregates regions into larger sets called *zones*, as explained in the next section.

1.2.3 Zones

Zones are particular polyhedra of $\mathbb{R}^{\mathcal{C}}_{\geq 0}$ definable by a constraint of $\Phi_{\mathcal{C}}$, that is, a set $Z \subseteq \mathbb{R}^{\mathcal{C}}_{\geq 0}$ such that there exists $g \in \Phi_{\mathcal{C}}$ such that $Z = [\![g]\!]$. Let us note that for a given zone Z, we do not have the unicity of g. For example, $x = y \wedge y \leq 1$ and $x = y \wedge x \leq 1$ represent the same zone. We can also note that, in the space of clocks, a zone is a polyhedron that is delimited by hyperplanes defined by $x = k$ or $x - y = k$, with $x, y \in \mathcal{C}$, $k \in \mathbb{Z}$.

We note a few basic operations defined on zones. First, the intersection $Z \cap Z'$ of two zones Z and Z' is clearly a zone. Given a zone Z, the set of time-successors of Z, defined as $Z\uparrow = \{v + t \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid t \in \mathbb{R}_{\geq 0},\ v \in Z\}$, is easily seen to be a zone; similarly for

Algorithm 1: Algorithm for reachability problem using regions

```
compute ⇒ from 𝒜 and M
wait = {(ℓ0, [u0]~)}
passed = ∅
while wait ≠ ∅ do
    take (ℓ, r) from wait
    if ℓ = ℓf ∧ r = rf then
        Stop and return true
    if (ℓ, r) ∉ passed then
        add (ℓ, r) to passed
        forall (ℓ', r') such that (ℓ, r) ⇒ (ℓ', r') do
            add (ℓ', r') to wait
return false
```

time-predecessors $Z{\downarrow} = \{v \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid \exists t \geq 0.\ v + t \in Z\}$. Given $R \subseteq \mathcal{C}$, we let $\mathsf{Reset}_R(Z)$ be the zone $\{v[R \leftarrow 0] \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid v \in Z\}$, and $\mathsf{Free}_x(Z) = \{v' \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid \exists v \in Z, d \in \mathbb{R}_{\geq 0}, v' = v[x \leftarrow d]\}$. We can also extend Free to a set of clocks R. We can define it by induction with $\mathsf{Free}_{x \cup R}(Z) = \mathsf{Free}_x(\mathsf{Free}_R(Z))$. We can define the successor of a zone by an edge as follows.

Definition 1.2.1. *For a zone Z, and an edge $e = (\ell, g, a, R, \ell')$, we denote $\mathsf{Post}_e(Z) = (\mathsf{Reset}_R(Z \wedge g) \wedge \mathsf{Inv}(\ell')){\uparrow} \wedge \mathsf{Inv}(\ell')$, that is the set of valuations reachable from a valuation of Z by taking the transition e and waiting some time in ℓ'.*

For the rest of this paper, we will only consider timed automata that have constraints of the form $x < k$ or $x \leq k$ in their invariants. This is a reasonable assumption as if this assumption is false for some timed automaton, we can build a timed automaton with the same set of reachable states by passing the constraints of an invariant $\mathsf{Inv}\ell$ to the guard of the edges going to ℓ, taking into account their reset. This is what is done in Uppaal or in [34].
With this assumption, we can write $\mathsf{Post}_e(Z) = \mathsf{Reset}_R(Z \wedge g){\uparrow} \wedge \mathsf{Inv}(\ell')$. By induction, we can extend the definition of Post_e to executions $\sigma = e_0....e_k e_{k+1}$ with $\mathsf{Post}_\sigma(Z) = \mathsf{Post}_{e_{k+1}}(\mathsf{Post}_{e_k}(...(\mathsf{Post} e_0(Z))))$.

Definition 1.2.2. *For a Z and an edge $e = (\ell, g, a, R, \ell')$, $\mathsf{Pre}_e(Z) = (\mathsf{Free}_R(Z wedge(r = 0)) \wedge g \wedge \mathsf{Inv}(\ell)){\downarrow}$, that is the set of valuations that can reach Z by waiting and taking the*

Figure 1.7 – For $e = (\ell, g, a, R, \ell')$ with $\mathsf{Inv}(\ell) = \mathsf{Inv}(\ell') = x \leq 6 \wedge y \leq 3$, $g = x \geq 1 \wedge y \geq 1$ and $R = \{y\}$, left figure represents in blue the successor of the gray zone by Post_e. On the right, the blue zone is the predecessor of the gray zone by Pre_e.

transition e.

From the definition we have:

Property 1. *For all zones Z, for all transitions $e = (\ell, g, a, R, \ell')$, for all valuations u': $u' \in \mathsf{Post}_e(Z)$ if and only if $\exists u \in Z, \exists d \in \mathbb{R}^+$ s.t. $(\ell, u) \xrightarrow{a}\xrightarrow{d} (\ell', u')$*

Proof. By definition of Post, $u' \in (Z \wedge g)[r \leftarrow 0]^{\uparrow} \wedge \mathsf{Inv}(\ell')$. So there is a $u'' \in (Z \wedge g)[r \leftarrow 0]$ such that $(\ell', u'') \xrightarrow{d} (\ell', u')$ and because $u'' \in (Z \wedge g)[r \leftarrow 0]$, there is a $u \in Z$ such that $(\ell, u) \xrightarrow{a} (\ell, u'')$.

If $(\ell, u) \xrightarrow{a} (\ell', u'') \xrightarrow{d} (\ell', u')$, then $u'' \in (Z \wedge g)[r \leftarrow 0]$ and $(Z \wedge g)[r \leftarrow 0]^{\uparrow} \wedge \mathsf{Inv}(\ell') = \mathsf{Post}_e(Z)$ □

Property 2. *For all zones Z, for all transitions $e = (\ell, g, a, r, \ell')$, for all valuations u $u \in \mathsf{Pre}_e(Z)$ iff $\exists u' \in Z, \exists d \in \mathbb{R}^+$ s.t. $(\ell, u) \xrightarrow{a}\xrightarrow{d} (\ell', u')$*

Proof. Similarly, we know that , $u \in (\mathsf{Free}_R(Z{\downarrow} \wedge (r = 0)) \wedge g \wedge \mathsf{Inv}(\ell))$. So there is a $u'' \in Z{\downarrow} \wedge (r = 0)$ such that $(\ell', u'') \xrightarrow{d} (\ell', u')$ and because $u'' \in Z{\downarrow} \wedge (r = 0)$, there is a $u \in (\mathsf{Free}_R(Z{\downarrow} \wedge (r = 0)) \wedge g \wedge \mathsf{Inv}(\ell))$ such that $(\ell, u) \xrightarrow{a} (\ell', u'')$.

If $(\ell, u) \xrightarrow{a} (\ell', u'') \xrightarrow{d} (\ell', u')$, then $u'' \in Z{\downarrow} \wedge (r = 0)$ and $u \in (\mathsf{Free}_R(Z{\downarrow} \wedge (r = 0)) \wedge g \wedge \mathsf{Inv}(\ell)) = \mathsf{Pre}_e(Z)$ □

We give an example of these successor and predecessor functions in Figure 1.7. Note that in general, $\mathsf{Pre}_e(\mathsf{Post}_e(Z)) \neq Z$

We can now build a transition system called the *zone graph* $\mathcal{T}_{zone}(\mathcal{A}) = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, \hookrightarrow)$ where the states (ℓ, Z) are pairs of a location and a zone, and the initial state is the initial location ℓ_0 coupled with $\vec{0}{\uparrow} \cap \mathsf{Inv}(\ell_0)$. The transition $\hookrightarrow$ is defined by $(\ell, Z) \overset{a}{\hookrightarrow} (\ell', Z')$

if and only if there is an edge $e = (\ell, g, a, R, \ell')$ such that $Z' = \mathsf{Post}_e(Z)$. We then have the following theorem.

Theorem 2. *For a timed automaton $\mathcal{A}$ with an initial state $\ell_0, \vec{0}$, for $\ell_f \in \mathcal{L}$, and u_f a clock valuation, $(\ell_0, \vec{0}) \to \cdots \to (\ell_f, u_f)$ in $\mathcal{T}(\mathcal{A})$ if and only if there exists a path $(\ell_0, \vec{0}\uparrow \cap \mathsf{Inv}(\ell_0)) \hookrightarrow \cdots \hookrightarrow (\ell_f, Z)$ with $u_f \in Z$ in the zone graph $\mathcal{T}_{zone}(\mathcal{A})$.*

So we have an infinite enumerable transition system and we will show in Section 1.2.5 how to limit ourselves to a finite number of zones. But first we will discuss the representation of these zones. Indeed, when we implement zones, we usually represent them with *Difference Bound Matrices* [24].

1.2.4 Difference Bound Matrices

First we need to represent atomic constraints. Let $\mathcal{C}_0 = \mathcal{C} \cup \{0\}$, where 0 is an extra symbol representing a special clock variable whose value is always 0. Then we can express atomic constraints as $x - y \lhd n$ where $x, y \in \mathcal{C}_0, n \in \mathbb{Z}, \lhd \in \{<, \leq\}$. Let $\mathbb{Z}_{\{<,\leq\}} = (\mathbb{Z} \times \{<, \leq\}) \cup \{(+\infty, <)\}$. We define the sum operation on this set as $(\leq, n_1) + (\leq, n_2) = (\leq, n_1 + n_2)$ and $\forall \lhd \in \{<, \leq\} (<, n_1) + (\lhd, n_2) = (<, n_1 + n_2)$. We also define the comparison $<$ on $\mathbb{Z}_{\{<,\leq\}}$ as follows. If $n_1 < n_2$ then $(\lhd, n_1) < (\lhd, n_2)$ and $(<, n) < (\leq, n)$.

A DBM is a $|\mathcal{C}_0| \times |\mathcal{C}_0|$-matrix taking values in $(\mathbb{Z} \times \{<, \leq\}) \cup \{(+\infty, <)\}$. Intuitively, cell (x, y) of a DBM D stores a pair $(d, \lhd)$ representing an upper bound on the difference $x - y$. For any DBM D, we let $[\![D]\!]$ denote the zone it defines.

While several DBMs can represent the same zone, each zone admits a *canonical* representation, also called reduced form or closed form, which is obtained by storing the tightest clock constraints defining the zone. This canonical representation can be obtained by computing shortest paths in a graph where the vertices are clocks and the edges weighted by clock constraints, with natural addition and comparison of elements of $(\mathbb{Z} \times \{<, \leq\}) \cup \{(+\infty, <)\}$. This graph has a negative cycle if, and only if, the associated DBM represents the empty zone. Algorithm 2 provides an implementation for this operation.

All the operations on zones can be performed efficiently (in $O(|\mathcal{C}_0|^3)$) on their associated DBMs while maintaining reduced form. For two DBMs M and N, we write $M \cap N$ for the reduced DBM describing $[\![M]\!] \cap [\![N]\!]$ and the intersection $N = D \cap D'$ of two canonical DBMs D and D' can be obtained by first computing the DBM $M = \min(D, D')$ such that

Algorithm 2: Close

```
1 Input:(D);
2 for k ∈ C_0 do
3     for i ∈ C_0 do
4         for j ∈ C_0 do
5             D_{i,j} :=
                min(D_{i,j}, D_{i,k} + D_{k,j})
```

Algorithm 3: Up

```
1 Input:(D);
2 for i ∈ C_0 do
3     D_{i,0} := (+∞, <)
```

Algorithm 4: Down

```
1 Input:(D);
2 for i ∈ C_0 do
3     D_{i,0} := (0, ≤) for j ∈ C do
4         if D_{j,i} < D_{0,i} then
5             D_{0,i} := D_{j,i}
```

Algorithm 5: Reset

```
1 Input:(D, R);
2 for x ∈ R do
3     for i ∈ C_0 do
4         D_{x,i} := D_{0,i}  D_{i,x} := D_{i,0}
```

Algorithm 6: Free

```
1 Input:(D, R);
2 for x ∈ R do
3     for i ∈ C_0, i ≠ x do
4         D_{x,i} := (+∞, <)  D_{i,x} := D_{i,0}
```

$M(x, y) = \min\{Z(x, y), Z'(x, y)\}$ for all $(x, y) \in {\mathcal{C}_0}^2$, and then turning M into canonical form.

We refer to [16] for full details. By a slight abuse of notation, we use the same notations for DBMs as for zones, writing e.g. $D' = D\uparrow$, where D and D' are reduced DBMs such that $[\![D']\!] = [\![D]\!]\uparrow$. This DBM can easily be computed by setting the cells $(x, 0)$ to $(+\infty, <)$ for all clocks x. This means that we keep only the diagonal constraints and the lower bound constraints. Similarly, we can define $D\downarrow$ as the the canonical DBM representing the zone $[\![D]\!]\downarrow$. Algorithms for these operations can be found in Algorithm 3 and Algorithm 2.

Given $R \subseteq \mathcal{C}$, we let $\mathsf{Reset}_R(D)$ be the reduced DBM that defines $\{v \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid \exists v' \in [\![D]\!], v = v'[R \leftarrow 0]\}$. This DBM can be computed by setting the cell $(x, 0)$ to $(0, \leq)$ for all clocks $x \in R$. This means that the upper constraints on the clocks of R have been reduced to 0. We can also define $\mathsf{Free}_R(D)$ as the reduced DBM that defines $\{v' \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid \exists v \in [\![D]\!], v = v'[R \leftarrow 0]\}$. We give some examples of the algorithms used to do such operations in Algorithm 5 and Algorithm 6. With all these operations we can also compute $\mathsf{Post}_e(D)$ and $\mathsf{Pre}_e(D)$ the canonical DBM representing the successor and the predecessor by e of the zone represented by D.

1.2.5 Extrapolation

Provisio. From here, we limit our study to timed automata whose guards and invariants contain no constraints of the type $x - y \triangleleft n$, $x, y \in \mathcal{C}, n \in \mathbb{Z}$.

For simplicity, we also consider only timed automata with no invariants, as for every timed automaton, we can build a timed automaton with no invariants that has the same transition system. If $\mathcal{A} = (\mathcal{L}, \ell_0, E, \mathsf{Inv})$, then the automaton $\mathcal{A}' = (\mathcal{L}, \ell_0, E', \mathsf{Inv}')$ where $\forall \ell \in \mathcal{L}, \mathsf{Inv}'(\ell) = true$ and $E' = \{(\ell, g \wedge \mathsf{Inv}(l), a, R, \ell') \mid (\ell, g, a, R, \ell') \in E\}$ is equivalent to $\mathcal{A}$ with respect to location reachability properties.

Maximum bounds abstraction

In order to limit ourselves to a finite system, we use the Maximum Bounds abstraction. Let M a clock ceiling. We denote by $\equiv_M$ the equivalence relation defined by $u \equiv_M v$ iff for all $x \in \mathcal{C}$, $u(x) = v(x)$ or $(u(x) > M(x)$ and $v(x) > M(x))$. Note that we already used this relation when we used regions. Indeed $u \sim_{reg,M} v$ if and only if $u \sim_{reg,M} v$ or $u \equiv_M v$. We then define an abstraction function α_M as $\alpha_M(Z) = \{u \mid \exists v \in Z, v \equiv_M u\}$. If M is chosen as the function that associates to every clock the maximum constant to which this clock is compared, then this abstraction is complete and sound with regards to reachability. This means that for a location ℓ, the set of locations reachable from (ℓ, Z) is exactly the set of location reachable from (ℓ, α_M). But this abstraction is not necessarily used in algorithms because α_M does not preserve the convexity of zones. The abstraction actually used is an extrapolation denoted by α_{Extra_M} such that for all zones Z we have $Z \subseteq \alpha_{Extra_M}(Z) \subseteq \alpha_M(Z)$ and $\alpha_{Extra_M}(Z)$ is a zone. From Z, we can compute $\alpha_{Extra_M}(Z)$ as follows:

— removing every atomic constraints $x - y < m$, and $x - y \leq m$ where $m > M(x)$.

— replacing all the constraints $x - y > m$ and $x - y \geq m$ where $m > M(x)$ by $x - y > M(x)$.

This is shown in Algorithm 7.

We define a new transition system with a transition $\mathcal{T}_{zone,M}(\mathcal{A}) = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, \hookrightarrow_{Extra_M})$ defined by $(\ell, Z) \overset{a}{\hookrightarrow}_{Extra_M} (\ell', Z')$ if and only if there is an edge $e = (\ell, g, a, R, \ell')$ such that $Z' = \alpha_{Extra_M} \mathsf{Post}_e(Z)$. Then we have from [16] [13]:

Theorem 3. *For a timed automaton $\mathcal{A}$ with initial state $(\ell_0, \vec{0})$ and no constraints of the type $x - y \triangleleft n$, for $\ell_f \in \mathcal{L}$ and Z_f a zone*

Algorithm 7: α_{Extra_M}

```
1 Input:(D, M);
2 for i ∈ C_0 do
3     for j ∈ C_0, j ≠ i do
4         if D_{i,j} > (M(i), ≤) then
5             D_{i,j} := (∞, <)
6         else if D_{i,j} < (−M(j), <) then
7             D_{i,j} := (−M(j), <)
```

— *If there is a path from* $(\ell_0, \vec{0}\uparrow)$ *to* (ℓ_f, Z_f) *in* $\mathcal{T}_{zone,M}(\mathcal{A})$ *then* (ℓ_f, u_f) *is reachable for all* $u_f \in Z_f$ *such that* $\forall x \in \mathcal{C}, u_f(x) \leq M(x)$.

— (ℓ_f, u_f) *for* u_f *such that* $\forall x \in C, u_f(x) \leq M(x)$, *implies that there is a path from* $(\ell_0, \vec{0}\uparrow)$ *to* (ℓ_f, Z_f) *in* $\mathcal{T}_{zone,M}(\mathcal{A})$ *such that* $u_f \in Z_f$.

Moreover, the transition system $\mathcal{T}_{zone,M}(\mathcal{A})$ is finite and can be built and explored easily. However, some improvements can be made.

Lower Upper bounds abstraction

In [13], a variant of this abstraction is used, where instead of one ceiling function, we use two functions L and U. We denote by $\prec_{L,U}$ the relation on clocks valuation defined by $u \prec_{L,U} v$ if and only if for all $x \in \mathcal{C}$, $v(x) = u(x)$ or $L(x) < u(x) < v(x)$ or $U(x) < v(x) < u(x)$. We can then define the abstraction function $\alpha_{L,U}$ as $\alpha_{L,U}(Z) = \{v \mid \exists u \in Z, u \prec_{L,U} v\}$. As above, if we choose $L(x)$ as the lowest value k such that $x > k$ or $x \geq k$ appears in a guard or invariant of the considered timed automaton, and $U(x)$ as the greatest value k such that $x < k$ or $x \leq k$ appears in a guard or invariant of the timed automaton. We have a sound and complete abstraction that does not preserve the convexity of zones, so we define an extrapolation $\alpha_{Extra_{L,U}}$ as :

— We remove all constraints $x - y < m$, and $x - y \leq m$ where $m > L(x)$.

— We replace all the constraints $x - y > m$ and $x - y \geq m$ where $m > U(x)$ by $x - y > U(x)$.

This can be seen in Algorithm 8. It was proven than for all zones Z and for all ceiling functions L, U, we have $Z \subseteq \alpha_{Extra_{L,U}}(Z) \subseteq \alpha_{L,U}(Z)$. We can then define the abstracted transition system defined by $\mathcal{T}_{zone,L,U}(\mathcal{A}) = (S, s_0, \Sigma \cup \mathbb{R}_{\geq 0}, \hookrightarrow_{Extra_{L,U}})$ and Theorem 3

Algorithm 8: $\alpha_{Extra_{L,U}}$

```
1 Input:(D, L, U);
2 for i ∈ C_0 do
3     for j ∈ C_0, j ≠ i do
4         if D_{i,j} > (L(i), ≤) then
5             D_{i,j} := (∞, <)
6         else if D_{i,j} < (−U(j), <) then
7             D_{i,j} := (−U(j), <)
```

holds for $\hookrightarrow_{Extra_{L,U}}$. This abstraction is a bit coarser but needs a deeper analysis of the automaton to find L and U.

Now that we have a finite transition system, we can modify Algorithm 1 to solve the reachability problem. The algorithm is very similar, but the abstraction we did aggregated the regions, which can result in fewer reachable states in the transition system. However, since each region is a zone, the number of zones can be greater than the number of regions, but if we look at zone inclusion, we can speed up the reachability analysis algorithm. Indeed, since we are only focused in reachability, we know that for two zones Z, Z' with $Z \subseteq Z'$, for a location ℓ, every location reachable from (ℓ, Z) is also reachable from (ℓ, Z'). So if the algorithm finds such a state (ℓ, Z) after it has computed the successors of (ℓ, Z'), then it is not necessary to compute the successors and start the exploration of (ℓ, Z). In Algorithm 9 we denote by $\mathcal{A}$ the automaton, (ℓ_0, Z_0) the initial state, ℓ_f the state we are trying to reach and Z_f the conditions on clocks we want to have when we reach ℓ_f.

The details of the implementation can be found in [16]. This algorithm is efficient, but it can be improved.

Diagonal constraints. For both extrapolations, we only gave the method if there are no constraints of the type $x - y \triangleleft n$ in the automaton. If there is that kind of constraints, there is a variant of these abstractions that can be found in [13]. Those abstraction, denoted by $\alpha^+_{Extra_M}$ and $\alpha^+_{Extra_{L,U}}$ are coarser, meaning that for a zone Z we have $Z \subseteq \alpha_{Extra_M}(Z) \subseteq \alpha^+_{Extra_M}(Z)$ and $Z \subseteq \alpha_{Extra_{L,U}}(Z) \subseteq \alpha^+_{Extra_{L,U}}(Z)$. Algorithms for such extrapolations are given in Algorithm 10 and Algorithm 11.

Lazy LU-abstraction. In [30], a method based on lazy abstraction for LU-abstraction is detailed. The main interest of this algorithm is that it used the precise LU-abstraction instead of the extrapolation to test the inclusion. The algorithm builds an *Adaptive*

Algorithm 9: Algorithm for reachability problem using zones

```
compute ↪ from 𝒜 and M (or L, U)
wait = {(ℓ0, Z0)}
passed = ∅
while wait ≠ ∅ do
    take (ℓ, Z) from wait
    if ℓ = ℓf ∧ Z ⊆ Zf then
        Stop and return true
    if Z ⊈ Z' for all (ℓ, Z') ∈ passed then
        add (ℓ, Z) to passed
        forall (ℓ', Z') such that (ℓ, Z) ↪_{Extra_M} (ℓ', Z') do
            add (ℓ', Z') to wait
return false
```

Algorithm 10: $\alpha^{+}_{Extra_M}$

```
Input:(D, M);
for i ∈ 𝒞0 do
    for j ∈ 𝒞0, j ≠ i do
        if D_{i,j} > (M(i), ≤) or D_{0,i} < (−M(i), <) or (D_{0,j} < (−M(j), <) and i ≠ 0)
         then
            D_{i,j} := (∞, <)
        else if D_{i,j} < (−M(j), <) and i = 0 then
            D_{i,j} := (−M(j), <)
```

Algorithm 11: $\alpha^{+}_{Extra_{L,U}}$

```
Input:(D, M);
for i ∈ 𝒞0 do
    for j ∈ 𝒞0, j ≠ i do
        if D_{i,j} > (L(i), ≤) or D_{0,i} < (−L(i), <) or (D_{0,j} < (−U(j), <) and i ≠ 0)
         then
            D_{i,j} := (∞, <)
        else if D_{i,j} < (−U(j), <) and i = 0 then
            D_{i,j} := (−U(j), <)
```

simulation graph. This is an exploration tree where the nodes are (q, Z, L, U) where q is a location, Z a zone and L, U are ceiling functions. The child (q', Z', L', U') of a node (q, Z, L, U) satisfies $Z' = \mathsf{Post}_e(Z)$ for $e = (l, g, a, r, l') \in E$. We stop exploring the child of a node (q, Z, L, U) if there is a node (q', Z', L', U') such that $Z \subseteq \alpha_{L',U'}(Z')$. Additionally, for each nodes, the ceiling functions L, U must satisfy some properties that we do not detail here. This is different from Algorithm 9 for three reasons.

— We have ceiling functions for each nodes, and for each node (q, Z, L, U), $L \geq L_{max}$, $U \geq U_{max}$ where L_{max}, U_{max} are the ceiling functions computed in the previous method. This means that $\alpha_{L,U}$ is coarser that $\alpha_{L_{max},U_{max}}$

— Here the nodes store Z and L, U. This allows us to use the abstraction $\alpha_{L,U}$ which is coarser than $\alpha_{Extra_{L,U}}$.

— The inclusion test here is $Z \subseteq \alpha_{L',U'}(Z')$. Indeed, it is shown in [29] that this test is practically as easy as testing $Z \subseteq Z'$, which is the test in Algorithm 9

Note that using local LU functions instead of the global LU function is something that is easy do even in Algorithm 9. Instead of using a global LU function, we can use LU functions that are specific to a location and based on the constraints on the location. We can also see that this algorithm is also based on refining the LU functions as we explore the adaptive simulation graph, hence the name adaptive.

We are also interested in using such methods, but on abstractions that are not necessary sound.

1.3 Counterexample Guided Abstraction Refinement

Abstraction is a process used extensively in model checking. Usually we have a model and we want to check if this model satisfies some properties. Since models can have a large number of states, the computation time can be too long, and abstraction is used to reduce the size of the model without losing the properties we want to check. An abstraction denotes an approximation of the model where states are merged, or more generally where the number of states to explore is reduced. This causes an over-approximation of the possible paths in the system. For example in Figure 1.8, we abstracted a simple red light by merging the states *yellow* and *red* in a *stop* state. But in the abstracted model, there is a path that cycles in *stop* and never reaches *go*, which is not possible in the concrete model.

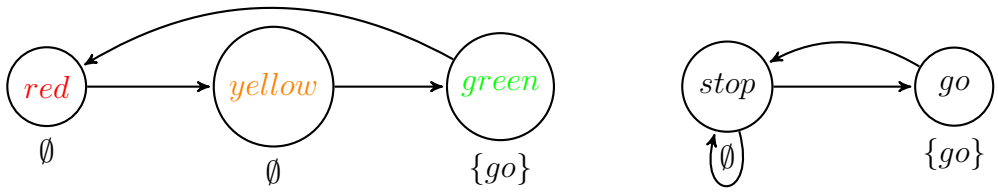


Figure 1.8 – An example of a Kripke structure for a traffic light M_{light} (on the left) and its abstraction M_{abs} (on the right).

In [27], a method called *CounterExample-Guided Abstraction Refinement* (CEGAR) is detailed. The general idea of a CEGAR loop, as shown in Figure 1.9 is to start with a coarse abstraction. This will create an over-approximation of the reachable states. This means that if a state is not reachable in our abstracted model, then it is not reachable in the concrete one either. On the other hand, if a state is reachable in the abstracted model, then we have an abstract path and we can check if this path is realizable in the concrete model. If that is the case, then we know that this state is reachable in the concrete model. Otherwise, we can use this counter-example to refine our abstraction, and have an abstraction where this abstract path does not exist. We can then restart the exploration of our new abstracted model and close the loop.

The CEGAR paradigm has been used already on many models, such as checking ACTL* formula on Kripke structures [21]. An improvement called *lazy abstraction* can be found in [27]. This paper describes an algorithm for automatic abstraction verification of C programs. Here the program is modeled with an abstract control flow graph, and when a spurious run is found, only the relevant part of the control flow graph is refined, so the rest of the control graph stays coarser and does not necessarily require further exploration. For real-time systems, it was used for collapsing locations on timed games in [25].

In [40], a CEGAR algorithm for timed automata can be found. The abstraction presented consists of removing clocks and the constraints of the automaton are projected on the remaining clocks, reducing the dimension of the valuation space we need to explore while possibly creating spurious behaviors in the abstracted model. With a comparison of the spurious run with the original model, one can detect the clocks needed, and add them back to the abstraction. Then starting with an abstraction that removes all clocks of a timed automaton, the CEGAR loop can be applied. This abstraction relies on the fact that some clocks are not needed, which is a strong assumption. One of the main problems is that, in practice, at the end of the CEGAR loop, we tend to obtain almost all, if not all

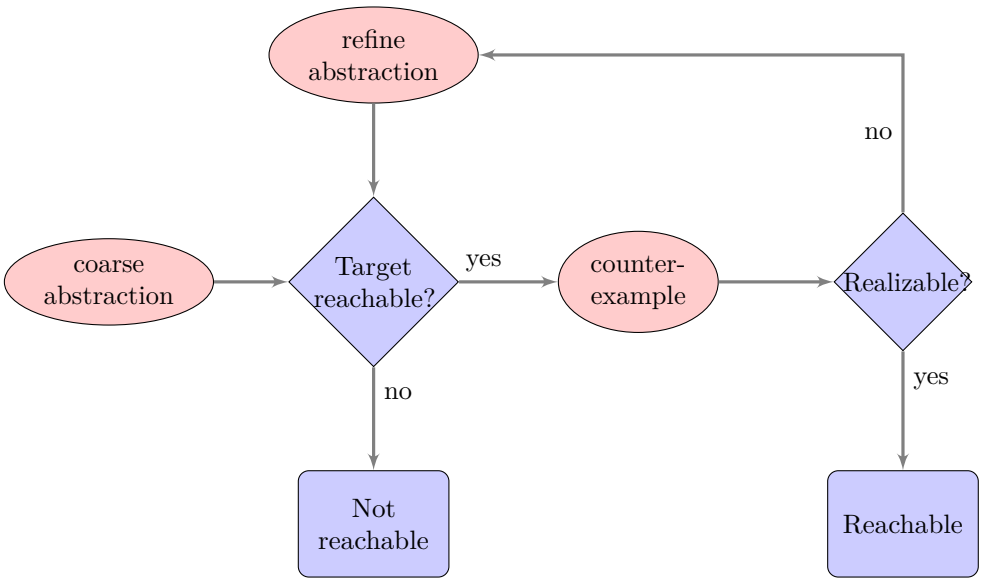


Figure 1.9 – Decision diagram for a CEGAR loop.

the clocks of the original automaton.

In [37] we can find once more the idea of studying a timed automaton with no clocks. More precisely, we remove the constraints on clocks, which essentially gives us a Kripke structure that is easier to study. Here the idea is to add new locations in the timed automaton when a constraint is needed. So if a run is realizable in the abstract model but not in the concrete model, it means that a transition $e = (\ell, g, R, \ell')$ cannot always be taken because we found a run where we could reach ℓ while not satisfying a constraint of g. In this case we duplicate the location ℓ into ℓ where this constraint is verified, and ℓ where this constraint is not verified. This is the abstraction refinement.

Some abstraction methods on timed automata are exact and as such, there is no need to use the CEGAR paradigm. The zones and regions abstraction, which are detailed in Section 1.2.5 are sound and complete, which means that no refinement is needed and that every property satisfied by the abstraction is satisfied by the concrete model, and vice versa. The main idea is to reduce the space of clocks valuation by separating it into "chunks" using an equivalence or a simulation relation. In [30], the method presented also uses lazy zone-based abstractions, which means that the abstraction function is defined differently for each discrete state, allowing for multiple local abstractions that are coarser than the global abstraction. This also allows us to refine only some of these local abstractions instead of a global abstraction. In [32] we can even find abstractions that do not preserve the zones, but are used just for the subsumption relation. We are also using a trace refinement

algorithm, similar to the one introduced in [47].

Chapter 2

Enumerative Algorithm

In this chapter we will describe enumerative algorithms that used Counter-Example Guided Abstraction Refinement to check the safety of timed automata. We implemented and evaluated these algorithms on benchmarks based on previously known timed automata, as well as new examples that we created for this purpose.

2.1 Abstraction refinement

The main idea is to create an abstraction based on the extrapolation we have seen before, but instead of using an overapproximation of the zones that is sound and complete for reachabilty, we use an abstraction that overapproximates the set of reachable locations, and use the CEGAR loop to refine these abstractions.

2.1.1 Quantization abstraction

Let $\mathcal{C}_0$ a set of clocks. If $\mathcal{D} = (\mathcal{D}_{x,y})_{x,y\in\mathcal{C}_0}$ is such that for all clocks x and y in $\mathcal{C}_0$, $\mathcal{D}_{x,y} \subseteq \mathbb{Z} \times \{\leq, <\}$ is a finite set, then $\mathcal{D}$ is an *abstract domain*. We call $\mathcal{D}$ the *concrete domain* if $\mathcal{D}_{x,y} = \mathbb{Z} \times \{\leq, <\}$ for all $x, y \in \mathcal{C}_0$.

A zone Z is $\mathcal{D}$-*definable* if it can be expressed as a conjunction of atomic constraints $x - y \triangleleft n$ with $x, y \in \mathcal{C}_0$ and $(n, \triangleleft) \in \mathcal{D}_{x,y}$. This is equivalent to saying that there exists a DBM D such that $Z = [\![D]\!]$ and $D(x, y) \in \mathcal{D}_{x,y} \cup \{\infty, <\}$ for all $x, y \in \mathcal{C}_0$. Note that we do not require this DBM D to be reduced as the reduction can introduce values that are not in $\mathcal{D}$. We also consider the empty zone $\emptyset$ to always be $\mathcal{D}$-definable, and by definition the zone $\mathbb{R}^{\mathcal{C}}_{\geq 0}$ is always $\mathcal{D}$-definable. We denote by $\mathcal{Z}_{\mathcal{C}}/\mathcal{D}$ the set of $\mathcal{D}$-definable zones.
For a domain $\mathcal{D}$, we denote by $\alpha_{\mathcal{D}} : \mathcal{Z}_{\mathcal{C}} \to \mathcal{Z}_{\mathcal{C}}/\mathcal{D}$ the abstraction function induced by $\mathcal{D}$, that associates with each zone Z its abstracted zone $\alpha_{\mathcal{D}}(Z)$, that is the smallest $\mathcal{D}$-definable zone that contains Z. Since $\mathbb{R}^{\mathcal{C}}_{\geq 0} \in \mathcal{Z}_{\mathcal{C}}/\mathcal{D}$ we know that there is at least one $\mathcal{D}$-definable

zone containing Z. Since $\mathcal{Z}_{\mathcal{C}}/\mathcal{D}$ is closed under intersection by definition, we know that if two $\mathcal{D}$-definable zones contain Z, then their intersection contains Z and is $\mathcal{D}$-definable. So we know that the smallest $\mathcal{D}$-definable zone that contains Z always exists and is unique, and $\alpha_{\mathcal{D}}$ is properly defined.

Formally, for a zone Z with $M = DBM(Z)$, then $\alpha_{\mathcal{D}}(Z) = [\![M']\!]$ such as $\forall x, y \in \mathcal{C}_0, M'_{x,y} = \min\{(k, \triangleleft) \in \mathcal{D}_{x,y} \mid M_{x,y} \leq (k, \triangleleft)\}$ with $\min \emptyset = (\infty, <)$.
We can observe that for some function $M : \mathcal{C} \to \mathbb{N}$ that we can extend to $\mathcal{C}_0$ with $M(0) = 0$, and with $\mathcal{D}_{x,y} = \{k \in \mathbb{Z} \mid -M(y) \geq k \geq M(x)\}$ for all $x, y \in \mathcal{C}_0$, then we have $\alpha_{\mathcal{D}} = Extra_M$. Similarly, for L, U two functions in $\mathbb{N}^{\mathcal{C}}$, extended to $\mathcal{C}_0$ and with $\mathcal{D}_{x,y} = \{k \in \mathbb{Z} \mid -U(y) \geq k \geq L(x)\}$ for all $x, y \in \mathcal{C}_0$, then we have $\alpha_{\mathcal{D}} = Extra_M$.
So our abstractions are similar to the extrapolation shown in Section 1.2.5, although those were sound and complete with regard to reachabilty for some well chosen M or L, U functions.

2.1.2 Interpolants

An *interpolant* for a pair of zones (Z_1, Z_2) with $Z_1 \cap Z_2 = \emptyset$ is a zone Z_3 with $Z_1 \subseteq Z_3$ and $Z_3 \cap Z_2 = \emptyset$ [1] [44]. Let us define the density of a DBM D as $d(D) = \#\{(x, y) \in {\mathcal{C}_0}^2 \mid D(x, y) \neq (\infty, <)\}$. An interpolant is simple if it can be represented by a DBM of density 1. We use interpolants to refine our abstractions, that is why in order not to add too many new constraints when refining, our aim is to find *minimal interpolants*. Notice that while any pair of disjoint convex polyhedra can be separated by hyperplanes, not all pairs of disjoint zones admit simple interpolants, this is because not all half-spaces delimited by hyperplanes are zones.

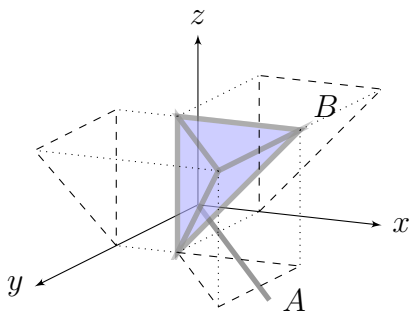
Lemma 1. *There exist pairs of zones accepting no simple interpolants.*

Proof. Consider 3-dimensional zones A, defined as $z = 0 \wedge x = y$, and B, defined as $y \geq 2 \wedge z \leq 2 \wedge y - x \leq 1 \wedge x - z \leq 1$. Both zones and their canonical DBMs are represented in Fig. 2.1.

We observe that they are disjoint: if a triple (x, y, z) were in both A and B, then $x = y$ and $z = 0$ (for being in B); in A, $y \geq 2$, hence also $x \geq 2$, contradicting $x - z \leq 1$.

Now, assume that there is a simple interpolant I, with $A \cap I = \emptyset$ and $B \subseteq I$. In the canonical DBM of I, only one non-diagonal element is not $(+\infty, <)$; assume

1. It is sometimes also required that the interpolant only involves clocks that have non-trivial constraints in both Z_1 and Z_2. We do not impose this requirement in our definition, but it will hold true in the interpolants computed by our algorithm.



$$A = \begin{pmatrix} (0,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \\ (\infty,<) & (0,\leq) & (0,\leq) & (\infty,<) \\ (\infty,<) & (0,\leq) & (0,\leq) & (\infty,<) \\ (0,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \end{pmatrix}$$

$$B = \begin{pmatrix} (0,\leq) & (3,\leq) & (4,\leq) & (2,\leq) \\ (-1,\leq) & (0,\leq) & (1,\leq) & (1,\leq) \\ (-2,\leq) & (1,\leq) & (0,\leq) & (0,\leq) \\ (0,\leq) & (1,\leq) & (2,\leq) & (0,\leq) \end{pmatrix}$$

Figure 2.1 – Two zones that cannot be separated by a simple interpolant

$I(x,y) \neq (+\infty,<)$. Then we must have $A(y,x) + I(x,y) < (0,\leq)$, and $B(x,y) \leq I(x,y)$. Then $A(x,y) + B(x,y) < (0,\leq)$. However, it can be observed that in our example, $A(x,y) + B(y,x) \geq (0,\leq)$ for all pairs (x,y). □

Still, we can bound the density of a minimal interpolant:

Lemma 2. *For any pair of disjoint, non-empty zones (A,B), there exists an interpolant of density less than or equal to $|\mathcal{C}_0|/2$.*

Proof. Assume that A and B are given as canonical DBMs, which we also write A and B for the sake of readability. We prove the stronger result that $A \cap B = \emptyset$ if, and only if, for some $n \leq |\mathcal{C}_0|/2$, there exists a sequence of pairwise-distinct clocks $(x_i)_{0\leq i\leq 2n-1}$ such that, writing $x_{2n} = x_0$,

$$\sum_{i=0}^{n-1} A(x_{2i}, x_{2i+1}) + B(x_{2i+1}, x_{2i+2}) < (0,\leq).$$

Before proving this result, we explain how we conclude the proof: the inequality above entails that

$$\bigcap_{i=0}^{n-1} A(x_{2i}, x_{2i+1}) \cap \bigcap_{i=0}^{n-1} B(x_{2i+1}, x_{2i+2}) = \emptyset$$

where we abusively identify $A(x,y)$ with the half-space it represents. It follows that $\bigcap_{i=0}^{n-1} B(x_{2i+1}, x_{2i+2})$ is an interpolant, whose density is less than or equal to $|\mathcal{C}_0|/2$.

Assume that such a sequence exists, and write C for the DBM $\min(A, B)$. Then

$$\begin{aligned}\sum_{i=0}^{2n-1} C(x_i, x_{i+1}) &= \sum_{i=0}^{2n-1} \min\{A(x_i, x_{i+1}), B(x_i, x_{i+1})\} \\ &\leq \sum_{i=0}^{n-1} A(x_{2i}, x_{2i+1}) + B(x_{2i+1}, x_{2i+2}) < (0, \leq).\end{aligned}$$

This entails that the intersection is empty.

Conversely, if the intersection is empty, then there is a sequence of clocks $(x_i)_{0 \leq i < m}$, with $m \leq |\mathcal{C}_0|$, such that, letting $x_m = x_0$, we have

$$\sum_{i=0}^{m-1} \min\{A(x_i, x_{i+1}), B(x_i, x_{i+1})\} < (0, \leq).$$

Consider one of the shortest such sequences. Since A and B are non-empty, the sum must involve at least one element of each DBM. Moreover, if it involves two consecutive elements of the same DBM (i.e., if $A(x_i, x_{i+1}) < B(x_i, x_{i+1})$ and $A(x_{i+1}, x_{i+2}) < B(x_{i+1}, x_{i+2})$ for some i), then by canonicity of the DBMs of A and B, we can drop clock x_{i+1} from the sequence and get a shorter sequence satisfying the same inequality, contradicting minimality of our sequence. The result follows. □

By adapting the algorithm of [44] for computing interpolants, we can compute minimal interpolants relatively efficiently: Computing a minimal interpolant can be performed in $O(|\mathcal{C}|^4)$.

Algorithm 12 describes our procedure. In order to prove its correctness, we begin with proving that the sequence of DBM it computes satisfies the following property:

Lemma 3. *For any $i \geq 0$ such that N^i has been computed by Algorithm 12, for any $(x, y) \in {\mathcal{C}_0}^2$, it holds*

$$N^i(x, y) = \min_{\substack{\pi \in \mathsf{Paths}(x,y) \\ |\pi|_B \leq i}} W_{\min(A,B)}(\pi).$$

Proof. The proof proceeds by induction on i. For $i = 0$, pick a path $\pi = (x_i)_{0 \leq i \leq k}$ from x_0 to x_k such that $|\pi|_B = 0$. Then

$$\begin{aligned}W_{\min(A,B)}(\pi) &= \sum_{0 \leq i < k} A(x_i, x_{i+1}) = \sum_{0 \leq i < k} M^0(x_i, x_{i+1}) \\ &= \sum_{0 \leq i < k} N^0(x_i, x_{i+1}) \geq N^0(x_0, x_k).\end{aligned}$$

Algorithm 12: Algorithm for minimal interpolant

```
Input:(canonical DBM A, B);
for (x, y) ∈ C_0^2 do
    if A(x, y) ≤ B(x, y) then
        M^0(x, y) := A(x, y);
    else
        M^0(x, y) := (∞, <);
N^0 := canonical(M^0);
for (i = 1; i ≤ |C_0|/2; i++) do
    for (x, y) ∈ C_0^2 do
        M^i(x, y) := min{N^{i-1}(x, y), min_{z∈SC_B(y)} N^{i-1}(x, z) + B(z, y)};
    for (x, y) ∈ C_0^2 do
        N^i(x, y) := min{M^i(x, y), min_{z∈SC_A(y)} M^i(x, z) + A(z, y)};
        if (x = y) and N^i(x, x) < (0, ≤) then
            return (true, i);
return false;
```

The first two equalities follow from the fact that π only involves transitions from x_i to x_{i+1} if $A(x_i, x_{i+1}) < B(x_i, x_{i+1})$; the third equality is because canonization will not modify entries from A (since A is originally in canonical form). The last inequality follows from canonicity of N^0.

Now assume that the result holds at step i, and that N^{i+1} is defined. Pick x and y in $\mathcal{C}_0$. By construction of M^{i+1} and N^{i+1}, there exists z and t in $\mathcal{C}_0$ such that $N^{i+1}(x, y) = N^i(x, z) + B(z, t) + A(t, y)$ with $t \in \mathsf{SC}_A(y)$ (or $t = y$) and $z \in \mathsf{SC}_B(t)$ (or $z = t$). From the induction hypothesis, there is a path π' from x to z such that $N^i(x, z) = W_{\min(A,B)}(\pi')$, and $|\pi'|_B \leq i$. Adding t and y to this path, we get a path π from x to y such that $N^{i+1}(x, y) = W_{\min(A,B)}(\pi)$ and $|\pi|_B \leq i + 1$.

It remains to prove that any path π from x to y with $|\pi|_B \leq i + 1$ is such that $N^{i+1}(x, y) \leq W_{\min(A,B)}(\pi)$. Fix such a path $\pi = (x_j)_{0 \leq j \leq k}$; we concentrate on the case where $|\pi|_B = i + 1$, since the other case follows from the induction hypothesis. We decompose π as $\pi_1 = (x_j)_{0 \leq j \leq l}$, $\pi_2 = (x_l, x_{l+1})$ and $\pi_3 = (x_j)_{l+1 \leq j \leq k}$, such that $(x_l, x_{l+1}) \in E_B$ and $(x_j, x_{j+1}) \in E_A$ for all $j > l$; in other terms, π_2 is the last E_B transition of π, and π_3 is a path from x_{l+1} to x_k only involving transitions in A. Then

$$w_{\min(A,B)}(\pi_2 \cdot \pi_3) = B(x_l, x_{l+1}) + w_A(\pi_3) \geq B(x_l, x_{l+1}) + w_A(x_{l+1}, x_k),$$

and

— if $(x_{l+1}, x_k) \in E_B$, then $w_{\min(A,B)}(\pi_2 \cdot \pi_3) \geq w_B(x_l, x_k)$, and, applying the induction hypothesis, $w_{\min(A,B)}(\pi) \geq N^i(x, x_l) + \min\{A(x_l, x_k), B(x_l, x_k)\}$. Since $M^{i+1}(x, x_l) \leq N^i(x, x_l)$, we get

$$\begin{aligned} w_{\min(A,B)}(\pi) &\geq \min\{N^i(x, x_l) + B(x_l, x_k), M^{i+1}(x, x_l) + A(x_l, x_k)\} \\ &\geq N^{i+1}(x, x_k). \end{aligned}$$

— if $(x_{l+1}, x_k) \in E_B$, then $w_{\min(A,B)}(\pi_2 \cdot \pi_3) \geq B(x_l, x_{l+1}) + A(x_{l+1}, x_k)$. From the induction hypothesis, $w_{\min(A,B)}(\pi) \geq N^i(x, x_l) + B(x_l, x_{l+1}) + A(x_{l+1}, x_k) \geq N^{i+1}(x, x_k)$.

□

Following the argument of the proof of Lemma 2, we get:

Corollary 1. *If $A \cap B \neq \emptyset$, then Algorithm 12 returns* `false`*; otherwise, it returns* $(\texttt{true}, k)$ *for the smallest k such that for all cyclic path π such that $|\pi|_B < k$, it holds* $w_{\min(A,B)}(\pi) \geq (0, \leq)$.

This entails that k is the dimension of the minimal interpolant. The minimal interpolant can be obtained by taking the B-elements of the negative cycle found by the algorithm.

2.2 Enumerative algorithm

In this section we present an algorithm based on an enumerative exploration. We will build and explore the zone graph in a similar fashion as Algorithm 9, but in this case we will apply an abstraction function before computing the successors function.

2.2.1 Core idea of the CEGAR loop

The main idea of this algorithm is to build an abstracted exploration tree. The nodes of this tree will contain a location ℓ, a zone Z and an abstract domain $\mathcal{D}$. In Algorithm 13, we show our implementation of the CEGAR loop. The initialization at line 1 chooses an abstract domain for the initial state, which can be either empty (thus being the coarsest abstraction) or defined according to some known data as we discuss in Section 2.2.3. The algorithm maintains the wait and passed lists that are used in the forward exploration. As usual, the wait list can be implemented as a stack, a queue, or another priority list

that determines the search order. The nodes of this wait list are called *unexplored nodes*. Conversely, nodes that are not in the waiting list are called *explored nodes* so that each node is either explored or unexplored. The algorithm also uses a relation of subsumption between nodes. Indeed if there are two node n and n' with $n.\ell = n'.\ell$, and $n'.z \subseteq \alpha_{n.\mathcal{D}}(n.Z)$, then we know that every location reachable from n' is also reachable from n. So exploring n and generating its abstract successors is enough and there is no need to explore the successors of n'. The algorithm explicitly creates an exploration tree that respects 4 properties:

P.1 The root of the tree is $n_0 = (\ell_0, \vec{0}\uparrow \cap \mathsf{Inv}(\ell_0), \mathcal{D}_0)$ for some domain $\mathcal{D}_0$

P.2 If a node $n = (\ell, Z, \mathcal{D})$ is covered by $n' = (\ell', Z', \mathcal{D}')$ then n' is not covered, $\ell = \ell'$ and $Z \subseteq \alpha'_{\mathcal{D}}(Z')$.

P.3 An unexplored node is not covered and has no children.

P.4 If an explored node $n = (\ell, Z, \mathcal{D})$ is not covered, then for every transition $e = (\ell, g, a, R, \ell')$ such that $\mathrm{Post}_e(\alpha_{\mathcal{D}}(Z)) \neq \emptyset$, n has a child $n' = (\ell', Z', \mathcal{D}')$ with $\mathrm{Post}_e(\alpha_{\mathcal{D}}(Z)) \subseteq Z'$

Line 2 creates a node containing location ℓ_0, zone $\vec{0}\uparrow \cap \mathsf{Inv}(\ell_0)$, and the abstract domain $\mathcal{D}_0$ as the root of our tree, and adds this to the wait list, thus ensuring that all the above properties are verified at the initialization of our algorithm. Procedure AbsExplore then looks for a trace to the target location ℓ_T. This procedure is detailed in Section 2.2.2. If such a trace exists, line 9 checks its feasibility. Here π is a sequence of node and edges of $\mathcal{A}$. The feasibility check is done by computing predecessors with zones starting from the final state. If the last zone intersects our initial zone, this means that the trace is feasible. More details are given in Section 2.2.3.

2.2.2 Abstract forward reachability: AbsExplore

We give a generic algorithm independently of the implementations of the abstraction functions and the refinement procedure.

Algorithm 14 describes the procedure that unfolds $\mathcal{A}$ into an exploration tree using an abstract successor function. It is similar to the standard forward reachability algorithm using a wait-list and a passed-list. In the resulting exploration tree the leaves are nodes in wait, covered nodes, or nodes that have no non-empty successors. Each node n contains the fields ℓ, Z which are labels describing the current location and zone; field covered points to a node covering the current node, field covering contains the list of nodes covered by the current node, field parent points to the parent node in the tree and field $\mathcal{D}$ is the abstract

Algorithm 13: Enumerative algorithm checking the reachability of a target location ℓ_T.

```
1  Input:(A = (L, Inv, ℓ0, C, E), ℓT)
     Initialize D0;
2  wait:= {node(ℓ0, 0⃗↑, D0)};
3  passed:= ∅;
4  while true do
5      π := AbsExplore(A, wait, passed, ℓT);

6      if π = ∅ then
7          return Not reachable;
8      else
9          if trace π is feasible then
10             return Reachable;
11         else
               Refine(π, wait, passed);
```

Algorithm 14: AbsExplore

```
1  Input:((L, Inv, l0, C, E), wait, passed, ℓT) while
     wait ≠ ∅ do
2      n := wait.pop();
3      if n.ℓ = ℓT then
4          return Trace from root to n;
5      if ∃n' ∈ passed such that n.ℓ =
         n'.ℓ ∧ n.Z ⊆ α_{n'.D}(n'.Z) then
6          n.covered := n';
7          add n to n'.covering;
8      else
9          for n' ∈ passed s.t.
             n.ℓ = n'.ℓ ∧ n'.Z ⊆ α_{n.D}(n.Z) do
10         n'.covered := n;
11         add n' to n.covering; remove n' from
           passed  passed.add(n);
12         for e = (ℓ, g, R, ℓ') ∈ E(n.ℓ) s.t.
             Z' := Post_e(α_{n.D}(n.Z)) ≠ ∅ do
13         D' := choose-dom(n, e);
14         n' := node(ℓ', Z', D');
15         n'.parent := n;
16         wait.add(n');
17 return ∅;
```

domain associated with the node. Thus, the algorithm uses a possibly different abstract domain for each node in the exploration tree. Also note that the field parent points to nothing only for the root, and that either the field covered or the field covering is left empty.

The difference of our algorithm w.r.t. the standard reachability can be seen at lines 9, 12 and 13. At line 12, we replaced the successor function with an abstract successor function, that is a composition of our abstraction function with a successor function. Similarly, when we test for covering at line 9, then we check if the current zone is covered by an abstracted zone.

At line 13, the function choose-dom chooses an abstract domain for the node n'. The abstraction function α depends on this domain, allowing us to have a lazy abstraction, where the abstraction might be coarser on some parts of the system. This also allows us to define variants for our abstraction function. In fact we will define two modes for our abstraction; one called *global domain* and the other called *local domain*.

Global domain means that we have a domain denoted $\mathcal{D}^g$ initialized with $\mathcal{D}^g_{x,y} = \{U(x), -L(y)\}$ with L and U the global lower bounds and upper bounds defined in Section 1.2.5. Then the choose-dom function will always return the current $\mathcal{D}^g$. Note that if a refinement occurs, new values might be passed to the domain $\mathcal{D}^g$ but all the nodes will not be updated with this new domain, so only nodes created after the refinement will have this refined domain. The advantage of this domain is that when the refinement procedure refines the abstract domain, all nodes benefit from this new information so the overall algorithm can converge faster in some cases. However, it is well-known that not all predicates are useful in every part of the search tree. So the abstraction in this variant can quickly become too refined which can render the abstraction-refinement loop inefficient.

Localized domain means that we have a function Δ that associates each location ℓ with a domain. Initially, for all ℓ we have $\Delta(\ell)_{x,y} = \{U_\ell(x), -L_\ell(y)\}$ with L_ℓ and U_ℓ the local lower bounds and upper bounds for ℓ defined in Section 1.2.5. Then the choose-dom(n) function returns $\Delta(n.\ell)$. This stems from the fact that not all states might need the same values and that we might want to have a coarser abstraction on some parts of our timed automata. Taking this a step further, in the case of a synchronized product of timed automata, our locations in $\mathcal{L}$ are actually tuples of $\mathcal{L}_0 \times \mathcal{L}_1 \times ... \times \mathcal{L}_n$. In this case, we can define a function δ that associates each location $\ell_i \in \mathcal{L}_0 \cup \mathcal{L}_1 \cup ... \cup \mathcal{L}_n$ with a domain, and we then have for $\ell = (\ell_0, \ell_1, ..., \ell_n) \in \mathcal{L}$, $\Delta(\ell)_{x,y} = \delta(\ell_0)_{x,y} \cup \delta(\ell_1)_{x,y} \cup ... \cup \delta(\ell_n)_{x,y} \cup \{U_\ell(x), -L_\ell(y)\}$. The motivation behind this choice can be seen for example in Figure 2.2. The non reachability of (ℓ_2, ℓ_6) is due to the fact that we cannot reach ℓ_1 with $x < 4$. So for $\ell \in \{\ell_3, \ell_4, \ell_5, \ell_6\}$ we want the domain of (ℓ_1, ℓ) to contain the constraint $x < 4$, in other words $(4, <) \in \delta(\ell_0)_{x,0}$

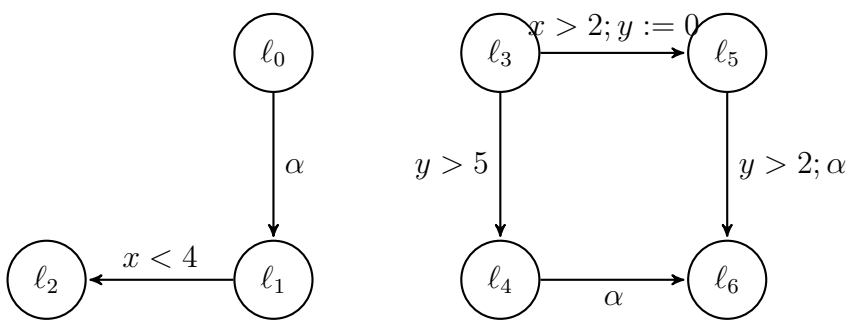


Figure 2.2 – An example of two timed automaton. In the synchronized product of those two on α, it can be seen that (ℓ_2, ℓ_6) is not reachable

Algorithm 15: Refine

```
1  Input:(π, wait, passed);
2  n := last node of π;
3  Z := n.Z;
4  r := Refine-rec(n, Z, wait, passed);
5  n_cut := node to cut
     (according to heuristics);
6  cut(n_cut);
7  if n_cut.Z = ∅ then
8  |  delete n_cut
9  else
10 |  passed.remove(n_cut);
11 |  wait.add(n_cut);
12 |  n_cut.Z := Concrete(n_cut);
13 return r;
```

Algorithm 16: Refine-rec

```
1  Input:(n, Z, wait, passed);
2  C := n.Z;
3  if C ∩ Z = ∅ then
4  |  Strengthen (n, Z, C, wait);
5  |  return Not Feasible;
6  else if n has no parent then
7  |  return Feasible ;
8  else
9  |  e edge from n.parent to n;
10 |  Z' := Pre_e(Z);
11 |  if Refine-rec(n.parent, Z', wait,
   |    passed)= Feasible then
12 |  |  return Feasible;
13 |  else
14 |  |  n.Z =
   |  |    Post_e(α_{n.parent.D}(n.parent.Z))
   |  |    C := n.Z;
15 |  |  Strengthen(n, Z, C, wait);
16 |  |  return Not Feasible;
```

Algorithm 17: Strengthen

```
1  Input:(n, Z, C, wait);
2  Update n.D if α_{n.D}(C) ∩ Z ≠ ∅ then
3  |  I :=interpolant(C, Z);
4  |  add(I) to domain;
5  Add every uncovered nodes to wait ;
```

As a first step towards proving correctness of our algorithm, we can make an easy observation about our algorithm AbsExplore:

Lemma 4. *Algorithm* AbsExplore *preserves Properties* ***P.1****,* ***P.2****,* ***P.3*** *and* ***P.4****.*

Note that although we use inclusion in Property **P.4**, AbsExplore would actually preserve equality of zones, but we will not always have equality before running AbsExplore. This is because Refine might change the zones of some nodes without updating the zones of all their descendants.

2.2.3 Refinement: Refine

We now describe our refinement procedure Refine. Let us now assume that AbsExplore returns $\pi = n_1 \xrightarrow{\sigma_1} n_2 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_{k-1}} n_k$, and write $\mathcal{D}_i$ for the updated domain of n_i and C_i for its zone. Note that we chose the updated domain, as we might have already done a refinement that showed this path was spurious before. We write A_i for the abstracted zone $\alpha_{\mathcal{D}_i}(C_i)$. Then π is not feasible if, and only if, $\mathsf{Post}_{\sigma_1 \dots \sigma_k}(C_1) = \emptyset$, or equivalently $\mathsf{Pre}_{\sigma_1 \dots \sigma_k}(A_k) \cap C_1 = \emptyset$. We note $Z_k = A_k$ and for $i < k$, $Z_i = \mathsf{Pre}_{\sigma_i}(Z_{i+1})$. Since for all $i < k$, it holds $\mathsf{Post}\sigma_i(A_i) \subseteq C_{i+1}$ by Property **P.4**. We can conclude that π is not feasible if, and only if, $\exists i \leq k.\ C_i \cap Z_i = \emptyset$. We illustrate this on Fig. 2.3.

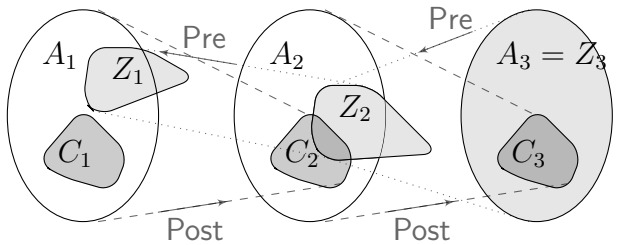


Figure 2.3 – Spurious counter-example: $Z_1 \cap C_1 = \emptyset$

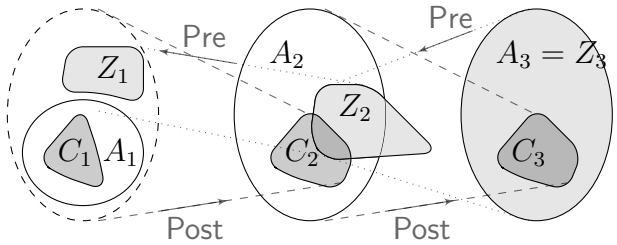


Figure 2.4 – Spurious counter-example: $Z_1 \cap A_1 = \emptyset$. The dashed area represent the previous value of A_1 used to compute the current value C_2.

Let us assume that π is not feasible. Let us denote by i_0 the maximal index such that $C_{i_0} \cap Z_{i_0} = \emptyset$. This index also has the property that for all $j < i_0$, we have $Z_j = \emptyset$ and $Z_{i_0} \neq \emptyset$. Once we have identified this trace as spurious by computing the Z_j, we have two possibilities:

— if $Z_{i_0} \cap A_{i_0} \neq \emptyset$: this means that we can reach A_k from A_{i_0} but not from C_{i_0}. In other words, our abstraction is too coarse and we must add some values to $\mathcal{D}_{i_0}$ so that $Z_{i_0} \cap A_{i_0} = \emptyset$. Those values are found by computing the interpolant of Z_{i_0} and C_{i_0}. This is the example shown in Figure 2.3.

— Otherwise it means that A_{i_0} cannot reach A_k and the only reason the trace exists is because $\mathcal{D}_{i_0}$ has been modified at some point after the computation of n_{i_0+1} and so

n_{i_0+1} has been computed as the successor of an abstracted zone computed with a coarser abstraction than the one we have now, and we need to update it. Such an example can be found in Figure 2.4

We can then update the values of C_i for $i > i_0$ and repeat the process until we reach an index j_0 such that $C_{j_0} = \emptyset$. We then have modified the nodes $n_{i_0}, \ldots, n_{j_0}$ and knowing that $n_{j_0}.Z = \emptyset$, we can delete it and all of its descendants. Since some of the descendants of n_{i_0} have not been modified, this might cause some refinements of the second type in the future. In order to ensure termination, we sometimes have to cut a subtree from a node in $n_{i_0}, \ldots, n_{j_0-1}$ and reinsert it in the wait list to restart the exploration from there. We call this action cut, and we can use several methods to decide when to use it. We considered 3 different methods that ensure termination:

1. `always-top`: we perform cut on the first node n_{i_0}. This is the same as restarting the exploration from the first node we modified, which can be seen as updating the subtree of this node.
2. `top-covered`: we perform cut on the first node n_{i_0} if it is covered by some other node. Since this node is covered, we know that we will not restart the exploration from this node, or that the node was covered by one of its descendant. If not, we delete n_{j_0} and its descendants.
3. `highest-covered`: we perform cut on the first node of $n_{i_0} ... n_{j_0}$ that is covered by some other node. If none of these nodes are covered, we delete n_{j_0} and its descendants.

Lemma 5. *Pick a node n, and let $Y = n.Z$. Then after running* Refine*, either node n is deleted, or it holds $n.Z \subseteq Y$. In other words, the zone of a node can only be reduced by* Refine*.*

Proof. Refine may only add values to the domain of a node, so that the refined zone is included in the previous one. If no values were added to the domain of a node, then its parent must have been modified. Since $A \subseteq B \Rightarrow \mathsf{Post}_e(A) \subseteq \mathsf{Post}_e(B)$, the result follows by induction. □

From this we get

Lemma 6. Refine *also preserves Properties **P.1**, **P.2**, **P.3** and **P.4**.*

Proof. — Property **P.1** is preserved since the root cannot be deleted.

— Property **P.2** is dealt with on line 5. Indeed we call Strengthen on every node that we modify. At the end of Strengthen, we update the covering for the node.

— The only node added to waiting is the node we applied cut on, so **P.3** is preserved.

— On line 14, we ensured that every node we modified still respect **P.4** with regards to their parents. And they still respect it regarding their children as either those children have been modified and have been ensured to respect **P.4**, or they have not been modified and using Lemma 5, we recognize that only the zone of the parent has been reduced, which preserves the property **P.4**

From this and Lemma 4 we can deduce:

Theorem 4. *Algorithm 13 satisfies Properties **P.1**, **P.2**, **P.3** and **P.4**.*

Corollary 2. *If* $\mathsf{wait} = \emptyset$ *then for every run* π' *starting from* $(\ell_0, \vec{0})$ *and finishing in* (ℓ, v) *in* $\mathcal{A}$*, there is a node* $n = (\ell, Z, \mathcal{D})$ *reachable in our exploration tree with* $v \in Z$*.*

Proof. This can be proven by induction on the length of the run π'. Indeed propert **P.1** ensures it for run of length 0. Otherwise ou run is $(\ell_0, \vec{0}) \xrightarrow{d_0, e_0} \cdots (\ell', v') \xrightarrow{d, e} (\ell, v)$ and by induction we know that there is a node $n' = (\ell', Z', \mathcal{D}')$ reachable in our exploration tree with $v' \in Z'$. If it is not covered, we note $n'' = n'$, otherwise we know that there is a covering node $n'' = (\ell', Z'', \mathcal{D}'')$ with $Z' \subseteq \alpha_{\mathcal{D}''}(Z'')$ from propert **P.2**, and so $v'A \in \alpha_{\mathcal{D}''}(Z'')$. In either cases, since n'' is not covered and since $v \in \mathsf{Post}_e(\alpha_{\mathcal{D}''}(Z'')) \neq \emptyset$, we know from property **P.4** that it has a children node $n = (\ell, Z, \mathcal{D})$ by e such that $(alpha_{\mathcal{D}''}(Z'') \subseteq Z$ and so $v \in Z$. □ □

We can then prove that our algorithm correctly decides the reachability problem and always terminates.

Theorem 5. *Algorithm 13 terminates and is correct.*

Proof. We first prove correctness, assuming termination. First let us notice that if AbsExplore($\mathcal{A}$, wait, passed, ℓ_T) returns $\emptyset$, then there is no node ℓ_T in our abstracted exploration tree and using Corollary 2, we know that ℓ_T is not reachable in $\mathcal{A}$. In other words, if the enumerative algorithm returns "*Not reachable*" then ℓ_T is indeed not reachable.

On the other hand, if the algorithm returns "*Reachable*", it means that there is a node $n_k = (\ell_T, Z, \mathcal{D})$ and AbsExplore returned $\pi = n_1 \xrightarrow{\sigma_1} n_2 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_{k-1}} n_k$ with $\mathsf{Pre}_{\sigma_1 \dots \sigma_k}(A_k) \cap (0\uparrow \cap \mathsf{Inv}(\ell_0)) \neq \emptyset$. From this we can deduce that $\pi = \sigma_1 \dots \sigma_k$ is a feasible trace reaching ℓ_T, and ℓ_T is indeed reachable.

We now prove termination. We will assume that we are in `highest-covered`, but the proof for `top-covered` is similar and the proof for `always-top` is easier. Since there are a finite number of possible locations and we can limit the number of possible zones to a finite number using abstraction functions, we can deduce that AbsExplore terminates.

Let us assume that the enumerative algorithm does not terminate. Then it means that Refine is called infinitely many times. Note that Refine is modifying a node and a node can be modified a finite number of time.

We can also note that a node can be destroyed only if one of its ancestors is modified. As such, we can show that for every depth k, there is a point in the algorithm where every node at depth k or less is fixed and will no longer be modified.

So we know that the algorithm does not terminate if, and only if, the depth of the resulting tree is unbounded. This means that there exists a path where we have two distinct nodes n_1 and n_2 with $n_1.\ell = n_2.\ell$ and $\alpha_{n_1.\mathcal{D}}(n_1.Z) = \alpha_{n_2.\mathcal{D}}(n_2.Z)$, since the number of location and possible zones is finite. Without loss of generality, we can assume that n_1 is an ancestor of n_2 and n_2 is the parent of another node. This is only possible if n_2 is in passed, which means that n_2 was in wait and was not covered at some point. Since the abstracted zone can only be modified to be smaller, this means that n_2 has been modified at any point. Otherwise n_2 has always been covered by n_1, which is not possible. Since n_2 has been modified, and it is covered (at least by n_1), this means that cut has been called on n_2 last time its zone has been modified. This means that n_2 has no children and is not in passed, contradicting our assumption. Hence the algorithm always terminates. □

2.2.4 Implementation discussion

Note that this algorithm requires to use the abstraction function every time we want to check the covering of a node, as we would have to check its zone against all the abstracted zone of zones sharing its location. Since the abstraction function is performed in $O(|\mathcal{C}_0|^2 \max(|\mathcal{C}_0|, \log_2(|\mathcal{D}|))$, this is something we would like to avoid. In reality, our implementation does not store the domain and the zone before abstraction for each node. Instead we only store one zone and use the abstraction function when the node is inserted in the passed list. This is because we want the node to contain the smallest zone possible when we test whether the node is covered. If it is not the case, then we can apply our abstraction and use the abstracted zone when we compute its successor and when we test whether the node is covering. This allows us to store a unique zone. Note that because of this we have to recompute the concrete successors for the Refine function. But since this

algorithm can be efficient only if we have a fairly short number of refinement, we prefer to have costly operations in Refine rather than in the successor and cover computation.

With that in mind, this algorithm has been implemented for TChecker with both the localized and globalized domain, as well as with all three variations `always-top`, `top-covered`, `highest-covered`. Moreover, you can see that this algorithm uses a passed and wait list, just like the algorithms presented in the first chapter. A good implementation of these lists results in faster executions, as it can facilitate the search for a subsumption relation. TChecker already has an efficient implementation of these lists as detailed in [30] [31].

We present experimental results for this algorithm and the one presented in the following chapter in Chapter 4.

Chapter 3

Symbolic Algorithm

In this chapter we present a symbolic algorithm, that reduces the reachability problem in timed automata to a problem of satisfiability for some Boolean formula. Indeed, this comes from the facts that a zone can be seen as small formula where the atomic constraints are the predicates. By encoding the state with a Boolean predicate, and seeing the successor operations as a succession of Boolean relations, we can build a Boolean formula that encodes the reachable part of the automaton.

3.1 Notation

We now present a symbolic algorithm that represents abstract states using Boolean formulas. Let $\mathbb{B} = \{0, 1\}$, and $\mathcal{V}$ be a set of variables. Let f a Boolean formula that uses variables from set $X \subseteq \mathcal{V}$, that is a combination of negations, conjunctions and disjunctions over the variables of X. We will write $f(X)$ to make the dependency explicit and sometimes write $f(X, Y)$ in place of $f(X \cup Y)$. Such a formula represents a set $[\![f]\!] = \{v \in \mathbb{B}^{\mathcal{V}} \mid v \models f\}$.

A *literal* is either p or $\neg p$ for a variable p. Given a set X of variables, a formula $f(X)$ is a X-*minterm* if it is the conjunction of literals where each variable of X appears exactly once. X-minterms can be seen as elements of $\mathbb{B}^X$ since they fix the value of each elements of X. For a Boolean formula on $F(X)$, we can denote $\mathsf{Minterms}(F)$ the set of X-minterms that satisfy F. Each of these minterm sis a valuation of $\mathbb{B}^X$ that satisfies $F(X)$.

For a formula $f(X)$, $g(X')$ and $x \in X$, the formula $f[g/x]$ is the *substitution of x by g in f*. It is a formula on $X \cup X'$ and is obtained by replacing each x with the formula $g(X')$. Given a vector of Boolean formulas $Y = (Y_x)_{x \in X}$, formula $f[Y/X]$ is the *substitution of X by Y in f*, obtained by replacing each $x \in X$ with the formula Y_x. The positive cofactor of $f(X)$ by x is $\exists x.\ (x \wedge f(X))$, and its negative cofactor is $\exists x.\ (\neg x \wedge f(X))$.

For a relation $R(X, X')$ and a formula $S(X, Y)$ representing a set of elements, let us define a generic operator `post` that computes the successors $S(X, Y)$ with the re-

lation $R(X, X')$ $\texttt{post}_R(S(X, Y)) = (\exists X.S(X, Y) \wedge R(X, X'))[X/X']$. Similarly, we set $\texttt{pre}_R(S(X, Y)) = (\exists X'.S(X, Y)[X'/X] \wedge R(X, X'))$, which computes the predecessors of $S(X, Y)$ by the relation R [36].

3.1.1 Clock predicate abstraction.

We fix a total order $\lhd$ on $\mathcal{C}_0$. In this section, abstract domains are defined as $\mathcal{D} = (\mathcal{D}_{x,y})_{x \lhd y \in \mathcal{C}_0}$, that is only for pairs $x \lhd y$. We can do so because constraints of the form $x - y \leq k$ with $x \rhd y$ can be encoded using the negation of $y - x < -k$ since $(x - y \leq k) \Leftrightarrow \neg(y - x < -k)$. So it is the same as using domains with $\mathcal{D}_{x,y} = -\mathcal{D}_{y,x}$.

For $x, y \in \mathcal{C}_0$, let $\mathcal{P}_{x,y}$ denote the set of *clock predicates associated to* $\mathcal{D}_{x,y}$:

$$\mathcal{P}^{\mathcal{D}}_{x,y} = \{P_{x-y \lhd k} \mid (k, \lhd) \in \mathcal{D}_{x,y}\}.$$

Let $\mathcal{P}^{\mathcal{D}} = \cup_{x,y \in \mathcal{C}_0} \mathcal{P}^{\mathcal{D}}_{x,y}$ denote the set of all clock predicates associated with $\mathcal{D}$ (we may omit the superscript $\mathcal{D}$ when it is clear). For all $(x, y) \in {\mathcal{C}_0}^2$ and $(k, \lhd) \in \mathcal{D}_{x,y}$, we denote by $p_{x-y \lhd k}$ the literal $P_{x-y \lhd k}$ if $x \lhd y$, and $\neg P_{y-x \lhd^{-1} -k}$ otherwise (where $\leq^{-1} = <$ and $<^{-1} = \leq$). So we can use $p_{x-y \lhd k}$ without consideration for the clocks order.

Here we have a strong justification for the use of these abstract domains. Most algorithms on Boolean formulas depend strongly on the number of Boolean predicates. Since the concrete domain creates too many predicates, this tends to slow down our algorithms. We also consider a set $\mathcal{B}$ of Boolean variables used to encode locations. Overall, the state space is described using Boolean formulas on these two types of variables, so states are elements of $\mathbb{B}^{\mathcal{P} \cup \mathcal{B}}$.

Our Boolean encoding of clock constraints and semantic operations follow those of [42] for a concrete domain. We define these however for abstract domains, and show how all successor computation and refinement operations can be performed.

Let us define the *clock semantics* of predicate $P_{x-y \lhd k}$ as $[\![P_{x-y \lhd k}]\!]_{\mathcal{C}_0} = \{\nu \in \mathbb{R}^{\mathcal{C}_0}_{\geq 0} \mid \nu(x) - \nu(y) \lhd k\}$. Since the set $\mathcal{C}$ of clocks is fixed, we may omit the subscript and just write $[\![P_{x-y \lhd k}]\!]$. We define the conjunction, disjunction, and negation as intersection, union, and complement, respectively. Given a $\mathcal{P}$-minterm $v \in \mathbb{B}^{\mathcal{P}}$, we define $[\![v]\!]_{\mathcal{D}} = \bigcap_{p \in v^{-1}(true)} [\![p]\!]_{\mathcal{D}} \cap \bigcap_{p \in v^{-1}(false)} [\![p]\!]^c_{\mathcal{D}}$. Thus, the negation of a predicate encodes its complement. For a Boolean formula $F(\mathcal{P})$, we set $[\![F]\!] = \bigcup_{v \in \mathsf{Minterms}(F)} [\![v]\!]_{\mathcal{D}}$. Intuitively, the minterms of $\mathcal{P}$ define smallest zones of $\mathbb{R}^{\mathcal{C}}_{\geq 0}$ definable using $\mathcal{P}$. Of course, if we used the concrete domain, those would be regions. For a domain $\mathcal{D}$, we call this set the $\mathcal{D}$-region. A minterm $v \in \mathbb{B}^{\mathcal{B} \cup \mathcal{P}}$

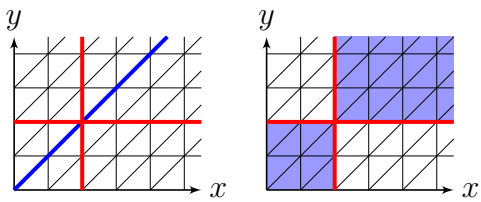


Figure 3.1 – On the left in blue the zone $x = y$ and on the right in blue its abstraction using the two predicates $x \leq 2, y \leq 2$ represented in red

defines a pair $[\![v]\!]_{\mathcal{D}} = (\ell, Z)$ where ℓ is encoded by $v_{|\mathcal{B}}$ and $Z = [\![v_{|\mathcal{P}}]\!]_{\mathcal{D}}$. A Boolean formula F on $\mathcal{B} \cup \mathcal{P}$ defines a set $[\![F]\!]_{\mathcal{D}} = \cup_{v \in \mathsf{Minterms}(F)} [\![v]\!]_{\mathcal{D}}$ of such pairs. A minterm v is *satisfiable* if $[\![v]\!]_{\mathcal{D}} \neq \emptyset$.

An abstract domain $\mathcal{D}$ induces an *abstraction function* $\alpha_{\mathcal{D}} \colon 2^{\mathbb{R}^{\mathcal{C}}_{\geq 0}} \to 2^{\mathbb{B}^{\mathcal{P}}}$ with $\alpha_{\mathcal{D}}(Z) = \{v \mid v \in \mathbb{B}^{\mathcal{P}} \text{ and } [\![v]\!]_{\mathcal{D}} \cap Z \neq \emptyset\}$, from the set of zones to the set of subsets of Boolean valuations on $\mathcal{P}$. We define the *concretization function* as $[\![\cdot]\!]_{\mathcal{D}} \colon 2^{\mathbb{B}^{\mathcal{P}}} \to 2^{\mathbb{R}^{\mathcal{C}}_{\geq 0}}$. The pair $(\alpha_{\mathcal{D}}, [\![\cdot]\!]_{\mathcal{D}})$ is a Galois connection, and $[\![\alpha_{\mathcal{D}}(Z)]\!]_{\mathcal{D}}$ is the most precise abstraction of Z in the domain induced by $\mathcal{D}$.

Note that although we use an abstract domain here, the abstraction function $\alpha_{\mathcal{D}}$ in this section is not the same as the one we defined in the previous section. This is due to the fact that here, we are not using zones but a Boolean formula that can define a union of zones. For example, $\alpha_{\mathcal{D}}$ is non-convex in general: if the clock predicates are $x \leq 2, y \leq 2$, then the set defined by the constraint $x = y$ maps to $(p_{x \leq 2} \wedge p_{y \leq 2}) \vee (\neg p_{x \leq 2} \wedge \neg p_{y \leq 2})$, which is not convex as shown in Figure 3.1.

3.1.2 Reduction

We now define the reduction operation, which is similar to the reduction of DBMs. The idea is to eliminate unsatisfiable minterms from a given Boolean formula. For example, we would like to make sure that in all minterms, if $p_{x-y \leq 1}$ holds, then so does $p_{x-y \leq 2}$, when both are available predicates. Another issue is to eliminate minterms that are unsatisfiable due to triangle inequality. This is similar to the shortest path computation used to turn DBMs in canonical form. We recall that reducing a DBM is the same as building a minimal graph where the nodes are the clocks and the weight are the constraints between those clocks.

Here we will do something similar, with a *path* in $\mathcal{D}$ being a sequence

$$x_1, (\alpha_1, \lhd_1), x_2, (\alpha_2, \lhd_2), \ldots, x_k, (\alpha_k, \lhd_k), x_{k+1}$$

where

$$x_1, \ldots, x_{k+1} \in \mathcal{C}_0$$

, and $(\alpha_i, \lhd_i) \in \mathcal{D}_{x_i, x_{i+1}}$ for $1 \leq i \leq k$. In other words, it is a path from clocks to clocks, where the weights are constraints of $\mathcal{D}$. Let us define $\mathsf{Paths}_k^{\mathcal{D}}(x - y \lhd \alpha)$ as the set of paths from x to y of length k and of weight at most $(\alpha, \lhd)$, that is, paths

$$x_1, (\alpha_1, \lhd_1), x_2, (\alpha_2, \lhd_2), \ldots, x_k, (\alpha_k, \lhd_k), x_{k+1}$$

with $x_1 = x$, $x_{k+1} = y$, and

$$\sum_{i=1}^{k} (\alpha_i, \lhd_i) \leq (\alpha, \lhd)$$

. We also denote

$$\mathsf{Paths}_{\leq k}^{\mathcal{D}}(x - y \lhd \alpha) = \bigcup_{l \leq k} \mathsf{Paths}_l^{\mathcal{D}}(x - y \lhd \alpha)$$

, meaning the paths from x to y of length at most k that weigh less than $(\lhd, \alpha)$. For a path $\pi = x_1, (\alpha_1, \lhd_1), \ldots, x_{k+1}$ and minterm v, let us write $v \models \pi$ for the statement $v \models \wedge_{i=1}^{k} p_{x_i - x_{i+1} \lhd \alpha_i}$.

Definition 3.1.1. *A minterm $v \in \mathbb{B}^{\mathcal{P}}$ is k-*reduced *if for all $(x, y) \in {\mathcal{C}_0}^2$ and $(\alpha, \lhd) \in \mathcal{D}_{x,y}$, for all $\pi = x_1, (\alpha_1, \lhd_1), x_2, \ldots, x_{k+1} \in \mathsf{Paths}_{\leq k}(x - y \lhd \alpha)$, whenever $v \models \pi$, we also have $v \models p_{x_1 - x_{k+1} \lhd \alpha}$ for all $(\alpha, \lhd) \in \mathcal{D}_{x_1, x_{k+1}}$ with $(\alpha, \lhd) \geq \sum_{i=1}^{k} (\alpha_i, \lhd_i)$.*

In other words, a minterm is k-reduced if and only if no contradictions can be obtained via paths of length less than or equal to k. Note that $|\mathcal{C}_0|$-reduction is equivalent to satisfiability since the condition then includes all the cycles, and it is known that in the absence of negative cycles, a set of difference constraints is satisfiable. Furthermore, for the concrete domain, k-reduction is equivalent to reduction for any $k \geq 2$. Indeed in a concrete domain we would have every predicate and his negation, so if a minterm is 2-reduced, it means that for all values $(\alpha_1, \lhd_1)$, $(\alpha_2, \lhd_2)$ and for every clock x, y, z, $v \models x - y \lhd_1 \alpha_1$ and $v \models y - z \lhd_2 \alpha_2$ implies $v \models x - z (\lhd_1 + \lhd_2)(\alpha_1 + \alpha_2)$. Since this is true for every possible value, this triangular inequality stands also for paths of lengths more than 2. A formula is said to be k-reduced if all its minterms are k-reduced.

Example 1. *Given predicates* $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}, p_{x-z\leq 2}\}$*, the formula* $p_{x-y\leq 1} \wedge p_{y-z\leq 1}$ *is the disjunction of the minterms* $p_{x-y\leq 1} \wedge p_{y-z\leq 1} \wedge p_{x-z\leq 2}$ *and* $p_{x-y\leq 1} \wedge p_{y-z\leq 1} \wedge \neg p_{x-z\leq 2}$*. But this last minterm is not satisfiable, therefore the formula is not 2-reduced. However, the same formula is reduced if* $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}\}$*, since then the formula contains only one minterm that is satisfiable.*

Consider now the predicate set $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}, p_{z-w\leq 3}, p_{x-w\leq 5}\}$*, and consider the formula* $\phi = p_{x-y\leq 1} \wedge p_{y-z\leq 1} \wedge p_{z-w\leq 3}$ *which is 2-reduced. Notice that* ϕ *does contain the minterm* $p_{x-y\leq 1} \wedge p_{y-z\leq 1} \wedge p_{z-w\leq 3} \wedge \neg p_{x-w\leq 5}$*. However,* ϕ *is 2-reduced since no path of length 2 allows to deduce* $p_{x-w\leq 5}$*. In the concrete domain,* $p_{x-y\leq 1} \wedge p_{y-z\leq 1}$ *would imply* $p_{x-z\leq 2}$*, thus,* $p_{x-w\leq 5}$ *could be derived too. So It is indeed because of the abstract domain that 2-reduction might fail to capture all shortest paths, since* $p_{x-z\leq 2}$ *is not a part of our abstract domain and as such cannot be used.*

We only consider 2-reduction since computing reductions is the most expensive operation in our algorithms, and the formula below defining 2-reduction already tends to grow in size. Let us define $\mathsf{reduce}^2_{\mathcal{D}}$ as follows

$$\bigwedge_{\substack{(x,y)\in\mathcal{C}_0{}^2\\(k,\lhd)\in\mathcal{D}_{x,y}}} \left[p_{x-y\lhd k} \leftarrow \left(\bigvee_{\substack{(l_1,\lhd_1)\in\mathcal{D}_{x,y}\\(l_1,\lhd_1)\leq(k,\lhd)}} p_{x-y\lhd_1 l_1} \vee \bigvee_{\substack{z\in\mathcal{C}_0,(l_1,\lhd_1)\in\mathcal{D}_{x,z},\\(l_2,\lhd_2)\in\mathcal{D}_{z,y}\\(l_1,\lhd_1)+(l_2,\lhd_2')\leq(k,\lhd)}} p_{x-z\lhd_1 l_1} \wedge p_{z-y\lhd_2 l_2} \right) \right]$$

The formula intuitively applies shortest paths over paths of length 1 or 2. In fact, for a formula S we note $\mathsf{reduce}^2_{\mathcal{D}}(S) = S \cap \mathsf{reduce}^2_{\mathcal{D}}$

Lemma 7. *For all formulas* $S(\mathcal{P})$*, we have* $[\![S]\!]_{\mathcal{D}} = [\![\mathsf{reduce}^2_{\mathcal{D}}(S)]\!]_{\mathcal{D}}$ *and all minterms of* $\mathsf{reduce}^2_{\mathcal{D}}(S)$ *are 2-reduced.*

Proof. It is easy to see that $[\![\mathsf{reduce}^k_{\mathcal{D}}(S)]\!]_{\mathcal{D}} \subseteq [\![S]\!]_{\mathcal{D}}$. In fact, any minterm of the former is a minterm of S as well by definition.

To see the converse, consider $v \in \mathsf{Minterms}(S)$. We show that $[\![v]\!]_{\mathcal{D}} \subseteq [\![\mathsf{reduce}^k_{\mathcal{D}}(S)]\!]_{\mathcal{D}}$. If $[\![v]\!]_{\mathcal{D}} = \emptyset$ then the inclusion holds trivially. Otherwise, we must have $v \in \mathsf{Minterms}(\mathsf{reduce}^k_{\mathcal{D}}(S))$. In fact, we show that all minterms $v \notin \mathsf{Minterms}(S)$ satisfy $[\![v]\!] = \emptyset$. Consider such a minterm v. There must exist $x, y \in \mathcal{C}_0$ and $(k, \lhd_k) \in \mathcal{D}_{x,y}$ such that $\neg p_{x-y\lhd k}$ but the right hand side of the implication holds. But this implies that $[\![v]\!] = \emptyset$.

The fact that all minterms of $\mathsf{reduce}^k_{\mathcal{D}}(S)$ are k-reduced follows by the definition of the operator. □

Example 2. *Note that* $\mathsf{reduce}^2_{\mathcal{D}}(S)$ *can still contain valuations that are unsatisfiable. Consider* $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}, p_{x-z\leq 4}, p_{z-x\leq -3}\}$*. Then the minterm* u *that sets all predicates to true is still contained in* $\mathsf{reduce}^2_{\mathcal{D}}(S)$ *although* $[\![u]\!] = \emptyset$ *since* $x - y \leq 1 \wedge y - z \leq 1$ *implies* $x - z \leq 2$ *which contradicts* $z - x \leq -3$*. Here, adding the predicate* $p_{x-z\leq 2}$ *or* $p_{x-z<3}$ *to the domain would make the reduction precise enough to remove this unsatisifable minterm in* $\mathsf{reduce}^2_{\mathcal{D}}(S)$*.*

We have already shown twice that adding predicates to the domain would refine the abstraction enough so that the reduced constraint eliminate a given unsatisfiable minterm. Here we will show how to do it.

Lemma 8. *Let* $v \in \mathbb{B}^{\mathcal{P}^{\mathcal{D}}}$ *be a minterm such that* $v \models \mathsf{reduce}^2_{\mathcal{D}}$ *and* $[\![v]\!] = \emptyset$*. One can compute in polynomial time a refinement* $\mathcal{D}' \supset \mathcal{D}$ *such that* $v \not\models \mathsf{reduce}^2_{\mathcal{D}'}$*.*

Proof. Consider a (non-canonial) DBM D that encodes v. Formally, for all $(x, y) \in {\mathcal{C}_0}^2$, $D(x, y) = \min\{(k, \triangleleft) \mid p_{x-y\triangleleft k} \in \mathcal{P}_{x,y} \text{ and } v(p_{x-y\triangleleft k}) = 1\}$. The corresponding graph must have a negative cycle $s_1 s_2 \ldots s_m$ with $s_m = s_1$. To define $\mathcal{D}'$, we add the following predicates to $\mathcal{D}$:

— $s_i - s_{i+1} \leq D(s_i, s_{i+1})$ for all $1 \leq j \leq m - 1$.

— $s_1 - s_j \leq D(s_1, s_2) + D(s_2, s_3) + \ldots + D(s_{j-1}, s_j)$ for $1 \leq j \leq m - 1$.

Intuitively, along the negative cycle, we are adding predicates to represent exactly each single step, and also each big step from s_1 to s_j. This allows to derive the negative cycle using only paths of length 2.

More precisely, $v \wedge \mathsf{reduce}^2_{\mathcal{D}'}$ implies the following two predicates: $s_1 - s_m \leq \sum_{j=1}^{m-1} D(s_j, s_{j+1})$, and $s_m - s_1 \leq D(s_m, s_1)$ as implied by v. Since $-D(s_m, s_1) > \sum_{j=1}^{m-1} D(s_j, s_{j+1})$ (due to the negative cycle), the first predicate entails that $\neg p_{s_1 - s_m \leq -D(s_m, s_1)}$, which contradicts the second one. Hence $v \not\models \mathsf{reduce}^2_{\mathcal{D}'}$. □

As you can see, this method for refining our domain is similar to the one defined to refine the domain in the previous chapter, by looking for a negative cycle in an intersection. So now we have a way of encoding our reachable zones and state using an abstract domain, a way of reducing this encoding dependent on this abstract domain and a way to refine this domain such that the reduction can remove a given unsatisfiable minterm. Now we need to be able to compute the successors.

3.2 Successor Computation

In this section, we explain how successor computation is realized in our encoding. For a guard g, assume we have computed an abstraction $\alpha_{\mathcal{D}}(g)$ in the present abstract domain. For each transition $\sigma = (\ell_1, g, R, \ell_2)$, let us define the formula $T_\sigma = \ell_1 \wedge \alpha_{\mathcal{D}}(g)$. We show how each basic operations on zones can be computed in our BDD encoding. In our algorithm, all formulas $A(\mathcal{B}, \mathcal{P})$ representing sets of states are assumed to be reduced, that is, $A(\mathcal{B}, \mathcal{P}) \implies \mathsf{reduce}^2_{\mathcal{D}}(A(\mathcal{B}, \mathcal{P}))$.

3.2.1 Intersection

The intersection operation is simply logical conjunction. In fact, given $A(\mathcal{B}, \mathcal{P})$ and $B(\mathcal{B}, \mathcal{P})$, we define $A(\mathcal{B}, \mathcal{P}) \sqcap B(\mathcal{B}, \mathcal{P}) = \mathsf{reduce}(A(\mathcal{B}, \mathcal{P}) \wedge B(\mathcal{B}, \mathcal{P}))$.

Lemma 9. *For all reduced formulas $A(\mathcal{P}), B(\mathcal{P})$, $A(\mathcal{P}) \wedge B(\mathcal{P}) = \alpha_{\mathcal{D}}([\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}})$.*

Proof. Consider $v \in \mathsf{Minterms}(A(\mathcal{P}) \wedge B(\mathcal{P}))$. Then $v \in \mathsf{Minterms}(A(\mathcal{P})) \cap \mathsf{Minterms}(B(\mathcal{P}))$. So $[\![v]\!] \subseteq [\![A(\mathcal{P})]\!] \cap [\![B(\mathcal{P})]\!]$, and $v = \alpha_{\mathcal{D}}([\![v]\!]) \subseteq \alpha_{\mathcal{D}}([\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}})$.

We will now show $\alpha_{\mathcal{D}}([\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}}) \subseteq A(\mathcal{P}) \wedge B(\mathcal{P})$, which is equivalent to $[\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}} \subseteq [\![A(\mathcal{P}) \wedge B(\mathcal{P})]\!]$. Consider any clock valuation ν in the LHS, and let $v = \alpha_{\mathcal{D}}(\nu)$. Since $\nu \in [\![A(\mathcal{P})]\!]_{\mathcal{D}}$ and $\nu \in [\![B(\mathcal{P})]\!]_{\mathcal{D}}$, we must have $v \in A(\mathcal{P}) \wedge B(\mathcal{P})$, so $\nu \in [\![A(\mathcal{P}) \wedge B(\mathcal{P})]\!]$. □

3.2.2 Time Successors

For the time successors, we define

$$S_{\mathsf{Up}} = \bigwedge_{\substack{x \in \mathcal{C} \\ (k,\lhd) \in \mathcal{D}_{x,0}}} (\neg p_{x-0\lhd k} \to \neg p'_{x-0\lhd k}) \bigwedge_{\substack{x,y \in \mathcal{C}_0, x \neq 0 \\ (k,\lhd) \in \mathcal{D}_{x,y}}} (p'_{x-y\lhd k} \leftrightarrow p_{x-y\lhd k}).$$

Note that this relation is not a function: for $x \leq 0, y = 0$, if $\neg p_{x-y\lhd k}$, then necessarily $\neg p'_{x-y\lhd k}$; but otherwise both truth values for $p'_{x-y\lhd k}$ are allowed. In fact, the formula only says that all lower bounds on clocks and diagonal constraints must be preserved. We let $\mathsf{Up}(A(\mathcal{B}, \mathcal{P})) = \mathsf{reduce}(\mathtt{post}_{S_{\mathsf{Up}}}(A(\mathcal{B}, \mathcal{P})))$. We can see that this operations similar to the time successor operation on the DBMs, where for a DBM D we remove the upper constraints $D_{x,0}$ for all clocks x while keeping all the other diagonals constraints. Of course

Figure 3.2 – Time successors: the successors of the gray zone on the left are the union of all blue dashed zones on the right.

here this operation is abstracted and depends on the abstract domain $\mathcal{D}$. In fact it provides an over approximation of the time successor function.

Example 3. *Figure 3.2 shows this operation applied to the gray zone on the left. The over-approximation is visible in this example. In fact, the blue dashed zone defined by* $Z = 3 < x < 5 \wedge 0 \leq y < 1 \wedge x - y < 4$ *(on the bottom right) is computed as a successor of the gray zone although no point of the gray zone can actually reach* Z *by a time delay. Adding more diagonal constraints to the abstract domain, for instance,* $x - y \leq 2$ *would refine the abstraction to remove some unreachable valuations.*

Lemma 10. *For any Boolean formula* $A(\mathcal{B}, \mathcal{P})$, $\alpha_{\mathcal{D}}([\![A]\!]\uparrow) \subseteq \mathsf{Up}(A)$. *Moreover, if* $\mathcal{D}$ *is the concrete domain and* A *is reduced, then this holds with equality.*

Proof. We show that $\alpha_{\mathcal{D}}([\![A]\!]\uparrow) \subseteq \mathsf{Up}(A)$, which is equivalent to $[\![A]\!]\uparrow \subseteq [\![\mathsf{Up}(A)]\!]_{\mathcal{D}}$. Let $\nu' \in [\![A]\!]\uparrow$, and let $\nu \in [\![A]\!]$ such that $\nu' = \nu + d$ for some $d \geq 0$. We write $u = \alpha_{\mathcal{D}}(\nu) \in \mathsf{Minterms}(A)$, and $v = \alpha_{\mathcal{D}}(\nu')$. We now show that $(u, v) \in S_{\mathsf{Up}}$. This suffices to prove the inclusion since $v \in \mathsf{Minterms}(\mathsf{Up}(A))$ implies that $\nu' \in [\![v]\!]_{\mathcal{D}} \subseteq [\![\mathsf{Up}(A)]\!]_{\mathcal{D}}$. Observe that $\alpha_{\mathcal{D}}(\nu)$ and $\alpha_{\mathcal{D}}(\nu')$ satisfy the same diagonal predicates of the form $p_{x-y \lhd \alpha}$ with $x, y \neq 0$, since $\nu' = \nu + d$. Moreover, any lower bound satisfied by u is also satified by v. Thus, $(u, v) \in S_{\mathsf{Up}}$.

Assume now that A is reduced. We show that $\mathsf{Up}(A) \subseteq \alpha_{\mathcal{D}}([\![A]\!]\uparrow)$. Let $v \in \mathsf{Minterms}(\mathsf{Up}(A))$, and let $(u, v) \in S_{\mathsf{Up}}$ with $u \in \mathsf{Minterms}(A)$. We claim that $[\![v]\!] \subseteq [\![u]\!]\uparrow$. This suffices to prove the inclusion since $\{v\} = \alpha_{\mathcal{D}}([\![v]\!]) \subseteq \alpha_{\mathcal{D}}([\![u]\!]\uparrow) \subseteq \alpha_{\mathcal{D}}([\![A]\!]\uparrow)$. To prove the claim, it suffices to see $[\![u]\!]$ and $[\![v]\!]$ as DBMs. In fact, S_{Up} precisely corresponds to the up operation on DBMs. In particular, u and v have the same diagonal constraints, and any lower bound of u is a lower bound of v. Because u is reduced, this implies that $[\![v]\!]\uparrow \subseteq [\![u]\!]$. □

Example 4. *Note that we do need that the domain is concrete to prove equality. In fact, assume the predicates are* $p_1 = y - x \leq 3$, $p_2 = x - y \leq 1$, $x \leq 1, y \leq 2, x \leq 2, y \leq 4$. *Then if* $u = x \leq 1 \wedge y \leq 2 \wedge p_1 \wedge p_2$ *and* $v = 1 \leq x \leq 2 \wedge y \leq 4 \wedge p_1 \wedge p_2$. *So* v *is larger than* $u{\uparrow}$.

Second, to see that the Up *operation can yield strict over-approximations, consider* $p_{x\leq 1} \wedge p_{y\geq 1}$ *in a domain with no predicate on* $x - y$. *The operator relaxes all upper bounds, yielding* $p_{y\geq 1}$ *which defines a set larger than the concrete time-successors* $y \geq 1 \wedge x - y \leq 0$.

Alternatively, we can also use the method described in [42, Theorem 2] to compute time successors, but the above relation will allow us to compute predecessors as well.

3.2.3 Reset

Last, we define the reset operation as follows. For any $z \in \mathcal{C}$,

$$
\begin{aligned}
S_{\mathsf{Reset}_z} = & \bigwedge_{(0,\leq)\leq(k,\lhd)\in\mathcal{D}_{z,0}} p'_{z-0\lhd k} \\
\wedge & \bigwedge_{\substack{x\neq z \\ (k,\lhd)\in\mathcal{D}_{x,z}}} \left(\bigvee_{\substack{(l,\lhd')\in\mathcal{D}_{x,0} \\ (l,\lhd')\leq(k,\lhd)}} p_{x-0\lhd' l} \right) \Rightarrow p'_{x-z\lhd k} \\
\wedge & \bigwedge_{\substack{y\neq z \\ (k,\lhd)\in\mathcal{D}_{z,y}}} \left(\bigvee_{\substack{(l,\lhd')\in\mathcal{D}_{0,y} \\ (l,\lhd')\leq(k,\lhd)}} p_{0-y\lhd' l} \right) \Rightarrow p'_{z-y\lhd k} \\
\wedge & \bigwedge_{\substack{y\neq z \\ (0,\leq)\leq(k,\lhd)\in\mathcal{D}_{z,y}}} p'_{z-y\lhd k} \\
\wedge & \bigwedge_{\substack{x,y\neq z \\ (k,\lhd)\in\mathcal{D}_{x,y}}} p'_{x-y\lhd k} \Leftrightarrow p_{x-y\lhd k}.
\end{aligned}
$$

Intuitively, the first conjunct ensures all non-negative upper bounds on the reset clock hold; the second conjunct ensures that a diagonal predicate $x - z \lhd k$ with $x \neq z$ is set to true if, and only if, an upper bound $(l, \lhd') \leq (k, \lhd)$ already holds on x. Once again this is similar to the reset we defined for DBMs, but for a DBM D, if z was reset, then we set $D_{x,z}$ to $D_{x,0}$ which was the tighter possible constraint on x if D was reduced. Of course here this is the tighter possible constraint that appears in the reduction relative to our domain $\mathcal{D}$. The third conjunct is symmetric to the second, and the last one ensures diagonals not affected by reset are unchanged. Let us define $\mathsf{Reset}_z(A) = \mathsf{reduce}(\mathtt{post}_{S_{\mathsf{Reset}_z}}(A))$.

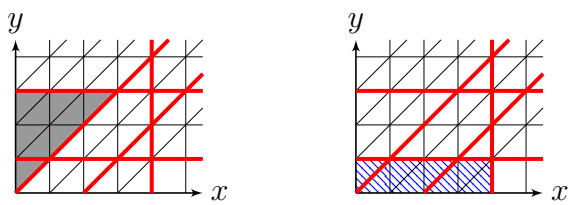


Figure 3.3 – Reset Operation: The abstract successor of the gray zone on the left is the blue dashed zone on the right.

Lemma 11. *For any Boolean formula $A(\mathcal{B}, \mathcal{P})$, and for any $z \in \mathcal{C}$, we have $\alpha_{\mathcal{D}}(\mathtt{Reset}_z([\![A]\!]_{\mathcal{D}})) \subseteq \mathsf{Reset}_z(A)$. Moreover, if $\mathcal{D}$ is the concrete domain, and A is reduced, then the above holds with equality.*

Proof. Let $r = \{z\}$. We show the equivalent inclusion $\mathtt{Reset}_r([\![A]\!]_{\mathcal{D}}) \subseteq [\![\mathsf{Reset}_r(A)]\!]_{\mathcal{D}}$. Let $\nu' \in \mathtt{Reset}_r([\![A]\!])$, and $\nu \in [\![A]\!]$ such that $\nu' = \nu[r \leftarrow 0]$. So $u = \alpha_{\mathcal{D}}(\nu) \in A$. Letting $v = \alpha_{\mathcal{D}}(\nu')$, let us show that $(u, v) \in S_{\mathsf{Reset}_r}$, which proves that $v \in \mathsf{Reset}_r(A)$, thus $\nu' \in [\![v]\!]_{\mathcal{D}} \subseteq [\![\mathsf{Reset}_r(A)]\!]_{\mathcal{D}}$.

— Consider $x \in r$. Since $\nu'(x) = 0$, we have that $v \models p_{x-y \lhd k}$ for all $(k, \lhd) \in \mathcal{D}_{x,y}$ with $(k, \lhd) \geq (0, \leq)$; so (u, v) satisfies the first conjunct. Fix $(k, \lhd) \in \mathcal{D}_{x,y}$.

— Assume $x \notin r, y \in r$, so that $\nu'(x) = \nu(x), \nu'(y) = 0$. If there is $(l, \lhd') \in \mathcal{D}_{x,0}$ with $(l, \lhd') \leq (k, \lhd)$, this means $\nu(x) - 0 \lhd' l \lhd k$ so $\nu'(x) - \nu'(y) \lhd k$. Thus the second conjunct is satisfied.

— Assume $x \in r, y \notin r$ so that $\nu'(x) = 0, \nu'(y) = \nu(y)$. If there is $(l, \lhd') \in \mathcal{D}_{0,y}$ with $(l, \lhd') \leq (k, \lhd)$, this means $0 - \nu(y) \lhd' l \lhd k$ so $\nu'(x) - \nu'(y) \lhd k$ as well. Thus the third conjunct is satisfied. We also have trivially $\nu'(x) - \nu'(y) \lhd k$ for all $(0, \leq) \leq (k, \lhd)$, which entails the fourth conjunct.

— Observe that ν and ν' satisfy the same predicates of type $p_{x-y \lhd \alpha}$ with $x, y \notin r$ since these values are not affected by the reset; so the pair (u, v) satisfies the last conjunct of S_{Reset_r} as well.

Now, if the domain is concrete and A is reduced, then we have $[\![\mathsf{Reset}_r(A)]\!]_{\mathcal{D}} \subseteq \mathtt{Reset}_r([\![A]\!]_{\mathcal{D}})$. In fact, the operation then corresponds precisely to the reset operation in DBMs of [16, Algorithm 10] as shown above. On DBMs, for a component (x, y) with $x \in r, y \notin r$, the algorithm consists in setting this component to value $(0, y)$. In our encoding, we thus set the predicate $p'_{x-y \lhd k}$ to true whenever $p_{0-x \lhd' l}$ holds with $(l, \lhd') \leq (k, \lhd)$. The argument is symmetric for (x, y) with $x \notin r$, $y \in r$. □

We now have all the components to compute our abstract successor operations.

3.3 Model-checking algorithm

3.3.1 Abstract successors computation

Algorithm 18 shows how to check the reachability of a target location given an abstract domain. The list layers contains, at position i, the set of states that are reachable in i steps. The function $\mathsf{ApplyEdges}$ computes the junction of immediate successors by all edges. It consists in looping over all edges $e = (l_1, g, R, l_2)$, and gathering the following image by e:

$$\mathsf{enc}(\ell_2) \wedge \mathsf{Reset}_{r_k}(\mathsf{Reset}_{r_{k-1}}(\ldots(\mathsf{Reset}_{r_1}((((\exists \mathcal{B}.A(\mathcal{B}, \mathcal{P}) \wedge \mathsf{enc}(\ell_1)) \wedge \alpha_{\mathcal{D}}(g)))))))),$$

where $R = \{r_1, \ldots, r_k\}$. We thus use a partitioned transition relation and do not compute the monolithic transition relation.

When the target location is found to be reachable, $\mathsf{ExtractTrace}(\mathsf{layers})$ returns a trace reaching the target location. This is standard and can be done by computing backwards from the last element of layers, by finding which edge can be applied to reach the current state. This is similar to what happens in the enumerative algorithm, where we compute the successive predecessors. Since both reset and time successor operations are defined using relations, predecessors in our abstract system can be easily computed using the operator $\mathtt{pre}_R$. As it is similar to what we already did in the previous chapter, we are not detailing this function here but assume that it returns a trace of the form

$$A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{n-1}} A_n,$$

where the $A_i(\mathcal{B}, \mathcal{P})$ are minterms and the σ_i belong to the trace alphabet $\Sigma = \{\mathrm{up}, r_\emptyset\} \cup \{r(x)\}_{x \in \mathcal{C}}$, with the following meaning:

— if $A_i \xrightarrow{\mathrm{up}} A_{i+1}$ then $A_{i+1} = \mathsf{Up}(A_i)$;

— if $A_i \xrightarrow{r_\emptyset} A_{i+1}$ then $A_{i+1} = A_i$;

— if $A_i \xrightarrow{r(x)} A_{i+1}$ then $A_{i+1} = \mathsf{Reset}_x(A_i)$.

The feasibility of such a trace is easily checked using DBMs.

The overall algorithm then follows a classical CEGAR scheme. We initialize $\mathcal{D}$ by adding the clock constraints that appear syntactically in $\mathcal{A}$, which is often a good heuristic.

Algorithm 18: Algorithm SymReach that checks the reachability of a target location l_T in a given abstract domain $\mathcal{D}$.

```
1  Input:(𝒜 = (ℒ, Inv, ℓ0, 𝒞, E), ℓT, 𝒟);
2  next := enc(l0) ∧ α_𝒟(∧_{x∈𝒞} x = 0);
3  layers := [];
4  reachable := false;
5  while (¬reachable ∧ next) ≠ false do
6      reachable := reachable ∨ next;
7      next := ApplyEdges(Up(next)) ∧ ¬reachable;
8      layers.push(next);
9      if (next ∧ enc(lT)) ≠ false then
10         return ExtractTrace (layers);
11 return Not reachable;
```

We run the reachability check of Algorithm 18. If no trace is found, then the target location is not reachable. If a trace is found, then we check for feasibility. If it is feasible, then the counterexample is confirmed. Otherwise, the trace is spurious and we run the refinement procedure described in the next subsection, and repeat the analysis.

3.3.2 Spurious trace detection

Since we initialize $\mathcal{D}$ with all clock constraints appearing in guards and invariants, we can make the following hypothesis.

Assumption 1. *All guards are represented exactly in the considered abstractions.*

Note that the algorithm can be easily extended to the general case; but this simplifies the presentation.

The abstract transition relation we use is not the most precise abstraction of the concrete transition relation. Therefore, it is possible to have abstract transitions $A_1 \xrightarrow{a} A_2$ for some action a while no concrete transition exists between $[\![A_1]\!]$ and $[\![A_2]\!]$. This requires care and is not a direct application of the standard refinement technique from [21]. A second difficulty is due to incomplete reduction of the predicates using $\mathsf{reduce}^2_{\mathcal{D}}$. In fact, some reachable states in our abstract model will be unsatisfiable. Let us explain how we refine the abstraction in each of these cases.

Consider an algorithm interp that returns an interpolant of two given zones Z_1, Z_2 as we have shown in the previous chapter. In what follows, by the *refinement of* $\mathcal{D}$

by $\mathsf{interp}(Z_1, Z_2)$, we mean the domain $\mathcal{D}'$ obtained by adding $(k, \triangleleft)$ to $\mathcal{D}_{x,y}$ for all constraints $x - y \triangleleft k$ of $\mathsf{interp}(Z_1, Z_2)$. Observe that $\alpha_{\mathcal{D}'}(Z_1) \cap \alpha_{\mathcal{D}'}(Z_2) = \emptyset$ in this case.

We define concrete successor and predecessor operations for the actions in Σ. For each $a \in \Sigma$, let Pre^c_a denote the concrete predecessor operation on zones defined straightforwardly, and similarly for Post^c_a.

Consider domain $\mathcal{D}$ and the induced abstraction function $\alpha_{\mathcal{D}}$. Assume that we are given a spurious trace $\pi = A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-1}} A_n$. Let $B_1 \ldots B_n$ be the sequence of concrete states visited along π in $\mathcal{A}$, that is, B_1 is the concrete initial state, and for all $2 \leq i \leq n$, let $B_i = \mathsf{Post}^c_{\pi_{i-1}}(B_{i-1})$. This sequence can be computed using DBMs.

The trace is *realizable* if $B_n \neq \emptyset$, in which case the counterexample is confirmed. Otherwise it is *spurious*. We show how to refine the abstraction to eliminate a spurious trace π.

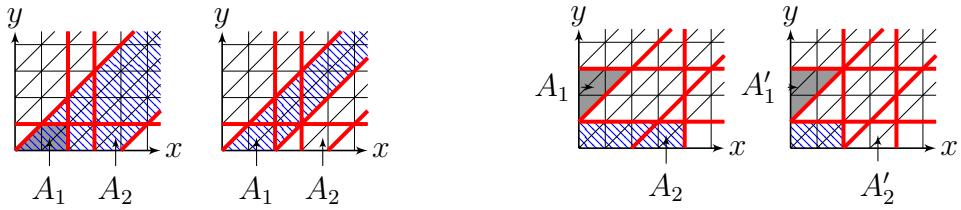
3.3.3 Refinement Procedure.

Let i_0 be the maximal index such that $B_{i_0} \neq \emptyset$. There are three possible reasons explaining why B_{i_0+1} is empty:

1. first, if the abstract successor A_{i_0+1} is unsatisfiable, that is, if it contains contradictory predicates; in this case, $[\![A_{i_0+1}]\!] = \emptyset$, and the abstraction is refined by Lemma 8 to eliminate this case by strengthening $\mathsf{reduce}^k_{\mathcal{D}}$.
2. if there are predecessors of A_{i_0+1} inside A_{i_0} but none of them are in B_{i_0}, i.e., $\mathsf{Pre}^c_{\pi_{i_0}}([\![A_{i_0+1}]\!]) \cap [\![A_{i_0}]\!] \neq \emptyset$; in this case, we refine the domain by separating these predecessors from the rest of A_{i_0} using $\mathsf{interp}(\mathsf{Pre}^c_{\pi_{i_0}}([\![A_{i_0+1}]\!]), B_{i_0-1})$, as in [21].
3. otherwise, there are no predecessors of A_{i_0+1} inside A_{i_0}: we refine the abstraction according to the type of the transition from step i_0 to $i_0 + 1$:
   (a) if $\pi_{i_0} = \text{up}$: refine $\mathcal{D}$ by $\mathsf{interp}([\![A_{i_0}]\!]\uparrow, [\![A_{i_0+1}]\!]\downarrow)$.
   (b) if $\pi_{i_0} = r(x)$: refine $\mathcal{D}$ by $\mathsf{interp}(\mathsf{Free}_x([\![A_{i_0}]\!]), \mathsf{Free}_x([\![A_{i_0+1}]\!]))$.

Note that the case $\pi_{i_0} = r_\emptyset$ is not possible since this induces the identity function both in the abstract and concrete systems.

Given abstraction $\alpha_{\mathcal{D}}$ and spurious trace π, let $\mathsf{refine}(\alpha_{\mathcal{D}}, \pi)$ denote the refined abstraction $\alpha_{\mathcal{D}'}$ obtained as described above. The following two lemmas justify the two subcases of the third case above. They prove that the detected spurious transition disappears after refinement. The reset and up operations depend on the abstraction, so we make



(a) Refinement for the time successors operation. The interpolant that separates $[\![A_1]\!]\uparrow$ from $[\![A_2]\!]\downarrow$ contains the constraint $x - y \leq 2$. When this is added to the abstract domain, the set A'_2 (which is A_2 in the new abstraction) is no longer reachable by the time successors operation.

(b) Refinement for the reset operation. The interpolant that separates $\mathsf{Free}_y(A_1)$ from $\mathsf{Free}_y(A_2)$ contains the constraint $x < 2$. When this is added to the abstract domain, the set A'_2 (which is A_2 in the new abstraction) is no longer reachable by the reset operation.

this dependence explicit below by using superscripts, as in Reset_x^α and Up^α, in order to distinguish the operations before and after a refinement.

Lemma 12. *Consider* $(A_1, A_2) \in \mathsf{Up}^\alpha$ *with* $[\![A_1]\!]\uparrow \cap [\![A_2]\!] = \emptyset$. *Then* $[\![A_1]\!]\uparrow \cap [\![A_2]\!]\downarrow = \emptyset$. *Moreover, if* α' *is obtained by refinement of* α *by* $\mathsf{interp}([\![A_1]\!]\uparrow, [\![A_2]\!]\downarrow)$, *then for all* $(A'_1, A'_2) \in \mathsf{Up}^{\alpha'}$, $[\![A'_1]\!] \subseteq [\![A_1]\!]$ *implies* $[\![A'_2]\!] \cap [\![A_2]\!] = \emptyset$.

Proof. Assume there is $v \in [\![A_1]\!]\uparrow \cap [\![A_2]\!]\downarrow$. There exists $d_1, d_2 \geq 0$ and $v_1 \in A_1$ such that $v = v_1 + d_1$ and $v + d_2 \in A_2$, which means $v_1 + d_1 + d_2 \in A_2$, thus $[\![A_1]\!]\uparrow \cap [\![A_2]\!] \neq \emptyset$.

Let $Z = [\![\mathsf{interp}([\![A_1]\!]\uparrow, [\![A_2]\!]\downarrow)]\!]$. By definition, $[\![A_1]\!]\uparrow \subseteq Z$, and $Z \cap [\![A_2]\!]\downarrow = \emptyset$. We have $Z\uparrow = Z$; in fact, Z cannot have upper bounds since $[\![A_1]\!]\uparrow$ does not have any. It follows that $Z\uparrow \cap [\![A_2]\!]\downarrow = \emptyset$.

Now, consider $(A'_1, A'_2) \in \mathsf{Up}^{\alpha'}$ with $[\![A'_1]\!] \subseteq [\![A_1]\!]$. Let us show that $[\![A'_2]\!] \subseteq Z$. Notice that all constraints of Z must be satisfied by A'_2 due to the inclusion $[\![A'_1]\!] \subseteq Z$: there are no upper bounds in Z, all its lower bounds are satisfied by A_1, thus also by A'_1, and are preserved by definition by Up, and all diagonal constraints of Z hold in A_1, thus also in A'_1, and are preserved as well in A'_2. This shows the required inclusion. It follows that $[\![A'_2]\!] \cap [\![A_2]\!] = \emptyset$. □

Lemma 13. *Consider* $x \in \mathcal{C}$, *and* $(A_1, A_2) \in \mathsf{Reset}_x^\alpha$ *such that* $[\![A_1]\!][x \leftarrow 0] \cap [\![A_2]\!] = \emptyset$. *Then* $\mathsf{Free}_x([\![A_1]\!]) \cap \mathsf{Free}_x([\![A_2]\!]) = \emptyset$. *Moreover, if* α' *is obtained by refinement of* α *by* $\mathsf{interp}(\mathsf{Free}_x([\![A_1]\!]), \mathsf{Free}_x([\![A_2]\!]))$, *then for all* $(A'_1, A'_2) \in \mathsf{Reset}_x^{\alpha'}$ *with* $[\![A'_1]\!] \subseteq [\![A_1]\!]$, *we have* $[\![A'_2]\!] \cap [\![A_2]\!] = \emptyset$.

Proof. Let $v \in \mathsf{Free}_x([\![A_1]\!]) \cap \mathsf{Free}_x([\![A_2]\!])$. Then there exist $v_1 \in [\![A_1]\!]$, $v_2 \in [\![A_2]\!]$ and v_0 such that $v_0 = v[x \leftarrow 0] = v_1[x \leftarrow 0] = v_2[x \leftarrow 0]$. But $[\![A_2]\!]$ is closed by resetting x, that is, $[\![A_2]\!][x := 0] \subseteq [\![A_2]\!]$. This follows from Lemma 11 applied to A_2, by observing that A_2 is unchanged by the reset operation. So $v_0 \in [\![A_2]\!]$. But then $[\![A_1]\!][x := 0] \cap [\![A_2]\!] \neq \emptyset$ as witnessed by $v_1[x := 0] = v_0$.

Let $Z = [\![\mathsf{interp}(\mathsf{Free}_x([\![A_1]\!]), \mathsf{Free}_x([\![A_2]\!]))]\!]$. For all A_1' satisfying $[\![A_1']\!] \subseteq [\![A_1]\!]$, we have $[\![A_1']\!][x \leftarrow 0] \subseteq [\![A_1]\!][x \leftarrow 0] \subseteq \mathsf{Free}_x([\![A_1]\!]) \subseteq Z$. So $Z \cap A_2 = \emptyset$ means that $[\![A_1']\!][x \leftarrow 0] \cap A_2 = \emptyset$. $\square$

Chapter 4

Experiments

In this chapter, we explain the models on which the algorithms presented in the previous chapters were evaluated, and then present the results of our tests on these models. We implemented the symbolic algorithm in OCaml in a tool called Symrob using the `CUDD` library and the OCaml wrapper `mlcuddidl`. The enumerative algorithm was implemented in C++ within an existing model checker called TChecker using Uppaal DBM library. It is worth noting that during this work, TChecker was redone completely. We reimplemented the enumerative algorithm, albeit a simpler version. Indeed, after each refinement, this simpler version restarts the exploration from the root, with a refined abstraction. Because of this, the results shown here are the ones from the original version of TChecker.

Both prototypes can take as input synchronized products of timed automata with invariants and discrete variables. It can also use a broader syntax of timed automata than the ones presented here, such as urgent and committed locations or weak synchronization. The presented algorithms are adapted to these features without difficulty.

We evaluated our algorithms on three classes of benchmarks that we developed and that we believe are significant. We also ran our tools on the CSMA/CD benchmarks with 3 to 12 processes, which is a protocol to ensure mutual exclusion. We compare the performance of our algorithms with that of Uppaal [15] which is based on zones, as well as the BDD-based model checker engine of PAT [38]. Here we will use the most recents algorithms presented in [39]. We were unable to compare with RED [46] which is not maintained anymore and not open source, and with which we failed to obtain correct results. The tool used in [26] was not available either. We thus only provide a comparison here with two well-maintained tools.

We will also note that these benchmarks were presented and made available during an artifact review in [41], and as such can still be used, since only small improvement have been made since.

4.1 Benchmarks

Two of our benchmarks are variants of schedulability-analysis problems where task execution times depend on the internal states of executed processes, so that an analysis of the state space is necessary to obtain a precise answer. All algorithms were given 8GiB of memory and a timeout of 30 minutes, and the experiments were run on laptop with an Intel i7@3.2Ghz processor running Linux. We ran the symbolic algorithm as well as the enumerative algorithm. For the enumerative algorithm, we implemented a version using a localized domain, as well as a version using a global domain. Moreover we implement 3 different ways to refine that we presented in Section 2.2.3: `always-top`, `top-covered` and `highest-covered`. This means that we have 6 different variations of the enumerative algorithm that we ran. Although we show the results for all 6 variants at the end of this chapter, in this section we will only consider the `highest-covered` variant for localized domain and the `always-top` variant for the global domain. This is mainly because representing all the 6 variant would make the following graphs hard to read, as well as the fact that on most examples, the difference between the 3 variants was too marginal to justify representing all 6 modes.

4.1.1 CSMA/CD

CSMA/CD is a fairly known protocol and a traditional benchmark for timed automata. It is a protocol used for collision detection when transmitting data and that has been used in Ethernet protocols for a long time. It is modeled by timed automata since the protocol solves collision by making other processes wait for some amount of time, as shown in [45]. The goal here is to verify the model to see if a state "collision" is reachable. We evaluate it for a different number of processes, and try to see how far it could scale for each algorithm.

The results in fig. 4.1 show that Uppaal performs the best but that our enumerative algorithm is slightly behind. The symbolic algorithm does not scale, probably due to the fact that adding clocks and diagonal constraints slows it down, while PAT fails to terminate in all cases. It is worth noting that PAT is slowed down if the constants used are high, which is not the cases for the other algorithms.

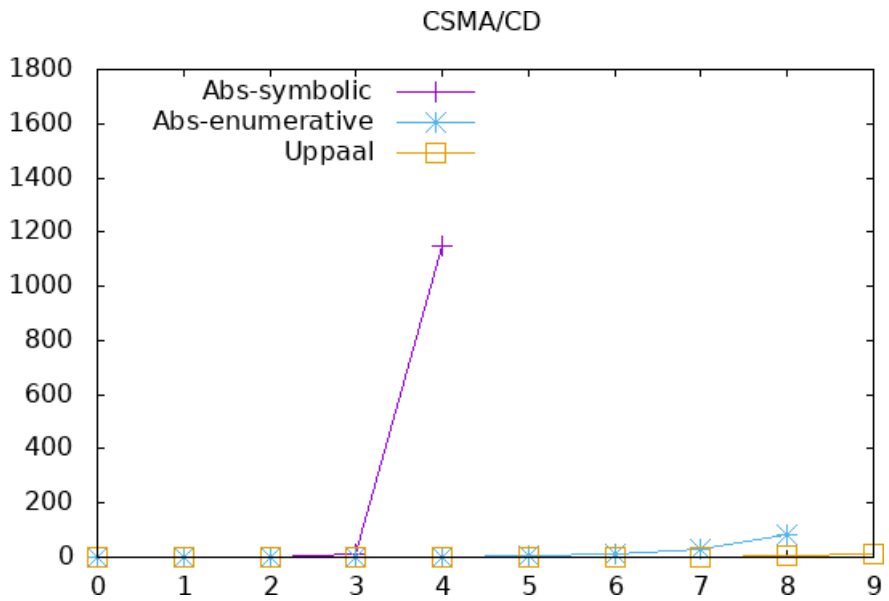


Figure 4.1 – Comparison of our enumerative and symbolic algorithms (L-enumerative, G-enumerative and Abs-symbolic) with Uppaal for CSMA. A point (X,Y) means X benchmarks were solved within time bound Y.

4.1.2 Monoprocess Scheduling Analysis

In this example, we build a model where a single process sequentially executes tasks on a single machine, and the execution time of each task depends on the state of the process. The goal is to determine a bound on the maximum execution time of a single cycle. This depends on the semantics of the process since the bound depends on the reachable states.

More precisely, we built a set of benchmarks where the processes are defined by synchronous circuit models taken from the Synthesis Competition [33]. In this circuit, latches represents resources being used by the system. We consider that changing the state of a resource takes some amount of time and so a subset of the latches have clocks associated with them, which measures the time elapsed since the latest value change. In other words, we measure the latest moment when the value changed from 0 to 1, or from 1 to 0. In our timed automata, a latch is represented as such: each latch is associated with positive bounds k_0 and k_1, and if the value of the latch changes from 0 to 1 (respectively from 1 to 0), then the execution time of the present task cannot be less than k_1 (respectively k_0). The execution time of the task is then defined as the minimum that satisfies these constraints. Modeling this as a timed automaton is done by creating a state that models the end of all the tasks. So we can verify the reachability of such a

state with a constraint on the global time, stating that such a state has to be reached in less than a given constant. If such a state is not reachable, we can answer that it is not possible to schedule these tasks within that time frame.

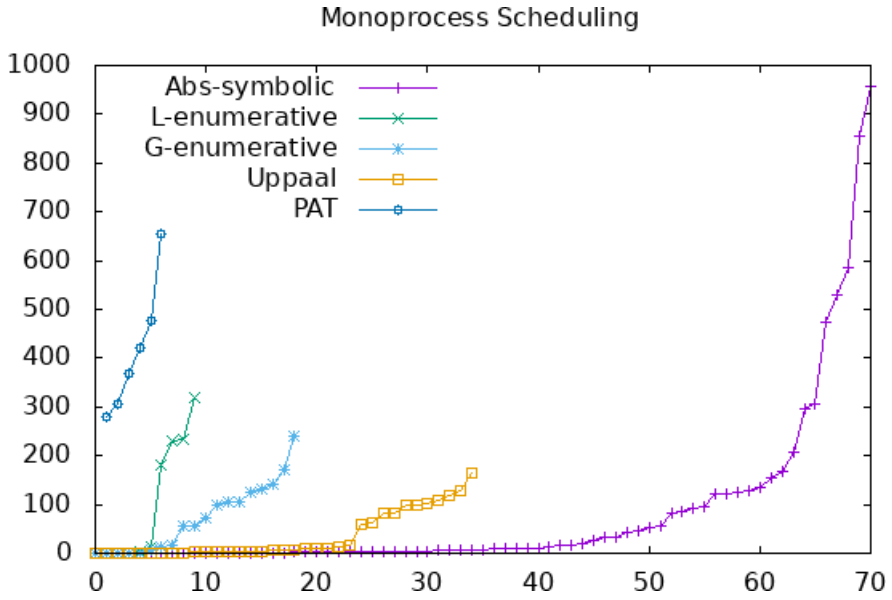


Figure 4.2 – Comparison of our enumerative and symbolic algorithms with Uppaal and PAT. for monoprocess scheduling analysis

The results of those tests are shown in fig. 4.2. The symbolic algorithm performs notably better than all the other algorithms on these examples. One of the possible reasons is that the Boolean encoding allows the algorithm to find the necessary refinements quickly in these benchmarks. The enumerative algorithm does not perform well on these examples, notably the localized version, due to a high number of refinements. On the other hand, the global version manages to find refinements for the entire model, avoiding unnecessary cycles of exploration and refinements. It is also worth noting that the symbolic algorithm initiates its domain with all the predicates that are present in the guards of the timed automaton, which might explain why it needs fewer refinements. Also, the enumerative algorithm tends to run out of memory on these benchmarks, which can indicate that the lazy approach is not the best in this case.

4.1.3 Multiprocess Stateful Scheduling Analysis

In this example, three processes are scheduled on two machines with a round-robin policy. Each of these processes schedules tasks one after the other without any delay.

Each task is described by a tuple (C_1, C_2, D) which defines the minimum and maximum execution times, and the deadline for the task. When a task finishes, the next task arrives immediately. The values in this tuple depend on the state of the process. The goal is to check the absence of any deadline miss. As in the previous benchmarks, a process executing a task (on any machine) corresponds to a step on a synchronous circuit model. Once again, we instantiate these circuits based on circuits from the Synthesis Competition.

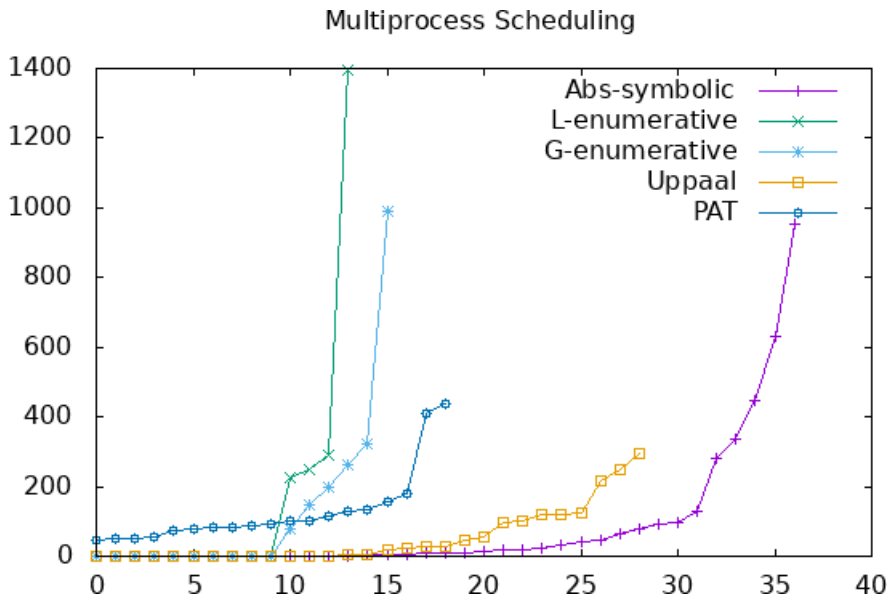


Figure 4.3 – Comparison of our enumerative and symbolic algorithms with Uppaal and PAT for multiprocess scheduling analysis

The results of those tests are shown in fig. 4.3 and are similar to the ones for monoprocess benchmarks, with a good performance of the symbolic algorithm. We note that once again the enumerative does not finish on a lot of these benchmarks, especially for the localized version.

4.1.4 Asynchronous Computation

This example is based on the notion of "threshold gates", defined as follows: each gate is characterized by a tuple $(n, \theta, [l, u])$ where n is the number of inputs, $0 \leq \theta \leq n$ is the threshold, and $l \leq u$ are lower and upper bounds on activation time. This gate has an output which is initially undefined, but then can take the value 0 or 1. After l units of times, the gate becomes activated and is then deactivated at u (or in other words it stays active for $u - l$ units of time). If all the inputs of a gate are defined, and if at least θ of

these inputs have value 1 while the gate is active, then it sets its output to 1. At the end of the time period, it becomes deactivated and the output becomes undefined again. Once deactivated, the gate restarts its loop, and wait l units of time, before it is activated again.

Let us assume a synchronized network of such gates. Our goal is to check whether a given gate can output 1 within a given time bound T. This is once again a model that can be modeled by timed automata, since every gate can be modeled by a location with constraint on an internal clock that is reset after u units of time.

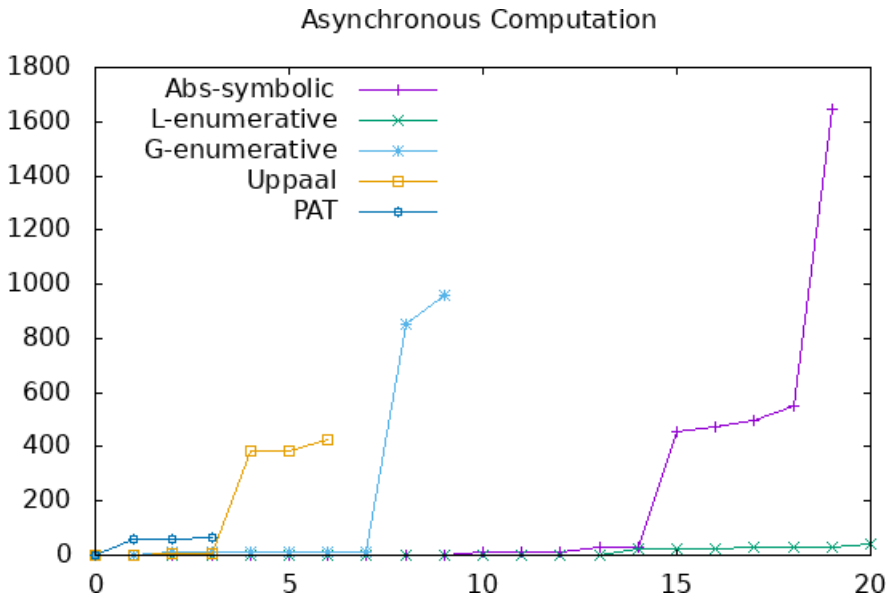


Figure 4.4 – Comparison of our enumerative and symbolic algorithms with Uppaal and PAT. for asynchronous waves

In fig. 4.4, we can see that the enumerative algorithm clears this benchmark easier than the other algorithms. This is because these benchmarks contain a lot of time constraints that are not always useful or relevant to the reachability problem considered. For Uppaal, these extra constraints would end up partitioning the space into many small zones and no zone would be covered by another. This partitioning is not always necessary to determine whether or not the accepting state can be reached, and as such would only slow down the traditional algorithms. The high number of constants on this algorithm could also explain why the symbolic algorithm is slower, since the symbolic algorithm already starts with every predicate that already appear in the timed automaton. This high number of initial predicates would only slow down this algorithm.

4.2 Overall results.

All of the results can be found in fig. 4.5, fig. 4.6, fig. 4.7 and fig. 4.9. In these tables we abreviated the `always-top`, `top-covered` and `highest-covered` by `cut`, `top` and `high` respectively. First of all, let us note that there are some cases where the enumerative algorithm does not finish the computation with some refinement modes, notably on the multiprocess and monoprocess scheduling, meaning that although these modes are equivalent on a lot of these examples, there are some cases where one is better than the other. Some explanations can be given on why some algorithms perform better than others. First of all, the shortcomings of the symbolic algorithm can be explained by its difficulty to scale. Indeed this algorithm is highly dependent on the number of predicates, which explains why it has some troubles with CSMA/CD. On the other hand, it manages to find the necessary refinements quickly in the monoprocess and multiprocess benchmarks and as such manages to finish quickly.

The enumerative algorithm does not perform well on these examples due to a high number of refinements, but manages to finish faster on the asynchronous wave benchmarks. Note that the high number of refinements can be limited by using a global domain, but that in the asynchronous wave benchmarks, the localized abstraction is better as it is better to have the coarsest abstraction possible in this case. We can also note that while these algorithms are used to answer whether or not a state is reachable, our algorithms are better suited for the cases when this state is not reachable. Indeed since checking if a state is reachable is the same as giving a path to this state, and since our algorithms are based on counter-example guided abstraction refinement, the only path that they could return is a feasible path. If this path is short, then Uppaal could have find this path easily, since these algorithms explore with a breadth-first search. If the path is long, then every predicate along the way needs to be present in our domain, which means that many refinements needs to have happened, which means that our algorithm has no chance to finish faster than Uppaal. This means that our algorithms are more adapted to safety problems, problems where we want to be sure than an error or unsafe state is not reachable. In this problem, we suppose that the state is not reachable in the model, and as such we suppose that our algorithms might finish faster than Uppaal. Of course we can also design other algorithms for problems where we want to prove that a some state is reachable and use the paths to this state. This is explained in the following chapter.

Name	Reachable	Uppaal	Symbolic	Ghig	Gtop	Gcut	Lhigh	Ltop	Lcut
csma-perso_10	False	11.11		28.97	29.1	29.8	28.56	28.57	27.38
csma-perso_11	False	22.08		88.88	89.14	89.07	79.97	79.97	78.27
csma-perso_12	False	121.2		264.99	264.86	264.77	243.34	237.39	232.42
csma-perso_2	False	0.0		0.02	0.03	0.02	0.02	0.02	0.03
csma-perso_3	False	0.0	1.91	0.03	0.02	0.03	0.03	0.02	0.02
csma-perso_4	False	0.0	0.23	0.04	0.04	0.04	0.04	0.04	0.04
csma-perso_5	False	0.01	0.43	0.09	0.09	0.09	0.12	0.11	0.09
csma-perso_6	False	0.02	10.61	0.25	0.25	0.27	0.28	0.28	0.23
csma-perso_7	False	0.08	1149.09	0.81	0.89	0.82	0.87	0.87	0.76
csma-perso_8	False	2.24		2.76	2.76	2.75	2.76	2.76	2.51
csma-perso_9	False	3.77		8.79	8.81	8.81	8.48	8.49	7.99
csma-perso_13	False	472.2					666.27	665.27	660.16

Figure 4.5 – Experimental results for CSMA/CD. For each timed automata, we show whether or not the target state was reachable, and how long it took for each algorithm to compute this result. If a cell has no results it means that the algorithm did not finish due to a lack of time or memory. PAT was not run on CSMA/CD due to its high dependencies to the constants.

Name	Reachable	Uppaal	PAT	Symbolic	Ghigh	Gtop	Gcut	Lhigh	Ltop	Lcut
a0_f	False			0.3	9.5	9.51	9.53	0.04	0.04	0.05
a1_f	False			2.29	959.42	957.02	959.1	0.09	0.09	0.09
a2_f	False			0.22	9.48	9.46	9.5	0.04	0.04	0.04
a7_f	False			0.2	9.52	9.51	9.52	0.04	0.04	0.05
a8_f	False			12.36	859.07	858.05	852.3	0.04	0.04	0.04
b0_150_t	True	0.0	0.8	0.03	0.04	0.04	0.04	0.04	0.04	0.05
b0_50_f	False	1.27	60.47	0.08	0.4	0.41	0.41	0.05	0.05	0.06
b1_150_f	False	387.58		1.31	9.62	9.62	9.61	0.04	0.04	0.05
b1_400_f	False	427.47		1646.9	9.6	9.6	9.59	0.04	0.04	0.05
b1_50_f	False	387.62		0.46	9.56	9.55	9.59	0.04	0.04	0.04
a10_f	False			473.12				0.94	0.93	0.95
a3_f	False			455.07				0.94	0.93	0.95
a5_f	False			11.73				29.71	29.63	29.79
a6_f	False			0.46				42.4	42.31	42.53
a9_f	False	4.81	61.18	28.22				21.19	21.16	21.4
b2_150_t	False							0.95	0.94	0.95
b2_50_f	False			499.48				0.94	0.94	0.97
b3_150_f	False			12.88				22.1	22.2	22.37
b3_300_t	False	4.44	63.92	30.96				22.12	22.22	22.38
b4_300_f	False							29.85	29.86	30.06
b5_300_f	False							29.71	29.7	29.94
a0	False			0.24						
a4_f	True			547.73						

Figure 4.6 – Experimental results for the asynchronous waves. For each timed automata, we show whether or not the target state was reachable, and how long it took for each algorithm to compute this result. If a cell has no results it means that the algorithm did not finish due to a lack of time or memory.

Name	Reachable	Uppaal	Symbolic	PAT
amba3b5y.aag__4L__200	True	9.05	1.25	278.10
amba3b5y.aag__4L__290	True	9.03	1.64	304.64
amba3b5y.aag__5L__290	True	10.9	2.49	420.20
amba3b5y.aag__6L__290	True	12.71	3.18	653.10
amba3b5y.aag__7L__290	True	16.88	8.54	
bs16y.aag__4L__100	True	0.01	0.12	
bs16y.aag__4L__150	False	0.01	0.19	
bs16y.aag__4L__200	False	0.01	0.2	
cnt5y.aag__4L__200	True	0.0	0.53	
cnt5y.aag__4L__300	False	0.0	0.54	
factory__assembly__3x3__1__1errors.aag__4L__200	True	4.28	0.91	
genbuf2b3unrealy.aag__4L__300	False	1.17	3.21	
genbuf2b3unrealy.aag__5L__250	True	0.13	1.53	
genbuf2b3unrealy.aag__5L__300	False	1.44	4.19	
genbuf2b3unrealy.aag__7L__400	False	1.65	2.51	
moving__obstacle__8x8__1glitches.aag__4L__150	True	4.31	0.33	
moving__obstacle__8x8__1glitches.aag__5L__150	True	5.05	0.52	
moving__obstacle__8x8__1glitches.aag__6L__150	True	5.87	1.43	
moving__obstacle__8x8__1glitches.aag__7L__150	True	7.15	3.97	
amba3b5y.aag__10L__300	False		956.55	
amba3b5y.aag__4L__300	False		1.24	369.36
amba3b5y.aag__5L__300	False		4.46	
amba3b5y.aag__6L__300	False		32.46	
amba3b5y.aag__7L__300	False		56.26	
amba3b5y.aag__8L__300	False		85.12	
amba3b5y.aag__9L__300	False		207.34	
amba4c7y.aag__10L__300	True		473.28	
amba4c7y.aag__4L__200	True	83.53	1.35	
amba4c7y.aag__4L__300	False		1.21	
amba4c7y.aag__5L__200	True	99.92	2.19	
amba4c7y.aag__5L__300	False		4.44	476.98
amba4c7y.aag__6L__300	True		17.25	
amba4c7y.aag__6L__500	False		12.54	
amba4c7y.aag__7L__300	True		32.52	
amba4c7y.aag__7L__500	False		121.7	
amba4c7y.aag__8L__300	True		53.48	
amba4c7y.aag__8L__500	False		167.3	
amba4c7y.aag__9L__300	True		136.51	
amba4c7y.aag__9L__500	False		296.39	
factory__assembly__3x3__1__1errors.aag__4L__300	False		0.81	
factory__assembly__3x3__1__1errors.aag__5L__300	False		3.34	
factory__assembly__3x3__1__1errors.aag__6L__300	True	101.12	5.67	
factory__assembly__3x3__1__1errors.aag__6L__400	False		11.41	
factory__assembly__3x3__1__1errors.aag__6L__500	False		9.72	
factory__assembly__3x3__1__1errors.aag__7L__500	False		93.69	
factory__assembly__3x3__1__1errors.aag__8L__500	False		126.42	
factory__assembly__3x3__1__1errors.aag__9L__500	False		531.07	
genbuf2b3unrealy.aag__10L__400	False	2.35	305.57	
genbuf2b3unrealy.aag__6L__300	False	1.63	1.49	
genbuf2b3unrealy.aag__7L__300	False	2.06	6.7	
genbuf2b3unrealy.aag__8L__400	False	2.04	6.0	
genbuf2b3unrealy.aag__9L__400	False	2.31	80.62	
genbuf5f5n.aag__10L__300	False		854.02	
genbuf5f5n.aag__5L__290	True	59.85	3.69	
genbuf5f5n.aag__5L__300	False		3.13	
genbuf5f5n.aag__6L__290	True	64.08	7.39	
genbuf5f5n.aag__6L__300	False		10.42	
genbuf5f5n.aag__7L__290	True	83.27	15.45	
genbuf5f5n.aag__7L__300	False		20.65	
genbuf5f5n.aag__8L__300	False		43.49	
genbuf5f5n.aag__9L__300	False		154.79	
moving__obstacle__8x8__1glitches.aag__10L__300	True	165.18	584.12	
moving__obstacle__8x8__1glitches.aag__4L__300	False		0.85	
moving__obstacle__8x8__1glitches.aag__5L__300	False		3.55	
moving__obstacle__8x8__1glitches.aag__6L__300	True	99.24	5.96	
moving__obstacle__8x8__1glitches.aag__6L__500	False		10.82	
moving__obstacle__8x8__1glitches.aag__7L__300	True	110.12	27.23	
moving__obstacle__8x8__1glitches.aag__7L__500	False		95.44	
moving__obstacle__8x8__1glitches.aag__8L__300	True	117.22	44.67	
moving__obstacle__8x8__1glitches.aag__8L__500	False		129.41	
moving__obstacle__8x8__1glitches.aag__9L__300	True	129.17	120.82	

Figure 4.7 – Experimental results for monoprocess schedulings for Uppaal, the symbolic algorithm and PAT.

Name	Reachable	Ghigh	Gtop	Gcut	Lhigh	Ltop	Lcut
amba3b5y.aag_4L_200	True	103.89	103.33	104.07			
amba3b5y.aag_4L_290	True	104.73	104.13	104.69			
amba3b5y.aag_5L_290	True	131.25	131.58	131.86			
amba3b5y.aag_6L_290	True	173.21	172.84	172.59			
amba3b5y.aag_7L_290	True	241.9	256.5	239.88			
bs16y.aag_4L_100	True	0.03	0.03	0.04	0.03	0.03	0.04
bs16y.aag_4L_150	False	0.04	0.04	0.03	0.44	0.42	0.23
bs16y.aag_4L_200	False	0.03	0.03	0.07	2.38	2.26	0.06
cnt5y.aag_4L_200	True	0.02	0.02	0.02	0.02	0.02	0.03
cnt5y.aag_4L_300	False	0.03	0.03	0.03	0.03	0.03	0.03
factory_assembly_3x3_1_1errors.aag_4L_200	True	54.91	55.02	55.09	234.52	236.14	
genbuf2b3unrealy.aag_4L_300	False	11.68	11.31	13.12			40.9
genbuf2b3unrealy.aag_5L_250	True	4.19	3.88	5.02	14.44	13.59	41.01
genbuf2b3unrealy.aag_5L_300	False	14.43	13.93	16.12			50.2
genbuf2b3unrealy.aag_7L_400	False	125.28	124.36	125.29			149.54
moving_obstacle_8x8_1glitches.aag_4L_150	True	56.19	54.84	55.25	182.29	171.74	244.06
moving_obstacle_8x8_1glitches.aag_5L_150	True	72.65	71.07	73.52	231.22	223.35	324.59
moving_obstacle_8x8_1glitches.aag_6L_150	True	96.97	94.16	97.52	320.14	305.09	448.63
moving_obstacle_8x8_1glitches.aag_7L_150	True	136.77	140.05	140.87			

Figure 4.8 – Experimental results for monoprocess schedulings with the enumerative algorithm.

Name	Reachable	Uppaal	PAT	Symbolic	Ghigh	Gtop	Gcut	Lhigh	Ltop	Lcut
1	True	0.01		0.43	0.03	0.03	0.07	0.03	0.03	0.03
10	True	0.0	77.53	0.04	0.02	0.02	0.04	0.02	0.02	0.03
15	True	0.0	55.2	0.04	0.02	0.02	0.02	0.03	0.02	0.03
17	False	28.0	81.34	12.89	248.82	248.69	262.08	291.81	290.14	293.45
18	True	0.0	52.55	0.05	0.02	0.02	0.02	0.02	0.02	0.02
19	False	21.26		8.89	191.98	192.14	196.54	225.12	231.28	231.98
20	True	0.01		0.43	0.03	0.03	0.06	0.04	0.04	0.03
22	True	0.01		0.42	0.03	0.03	0.11	0.02	0.02	0.02
25	True	0.0	100.5	0.06	0.03	0.02	0.04	0.03	0.02	0.02
27	False	28.12	158.35	1.31	278.53	307.21	324.33			
28	True	0.0	116.07	0.09	0.03	0.02	0.05	0.02	0.02	0.03
29	False	4.06		3.2	875.98	915.99	989.61	1394.06		1427.22
30	False	6.29	179.1	0.95	66.45	73.42	79.16			
31	True	0.0	128.78	0.12	0.02	0.02	0.03	0.02	0.02	0.02
32	False	0.96		2.46	133.71	143.84	147.98	246.68	346.84	252.22
34	True	0.0	409.25	0.07	0.03	0.03	0.03	0.03	0.03	0.04
11	False	218.7		46.62						
12	False	293.95	91.28	93.45						
14	False	121.35	87.28	32.84						
16	False	100.75		20.34						
24	False	117.79	135.26	3.42						
26	False	17.33		8.61						
33	False		439.3	129.72						
35	False			96.58						
36	False			631.87						
37	False			79.65						
38	False			281.21						
39	False			448.91						
3	False	56.35	72.08	20.1						
40	False			952.91						
41	False			337.0						
4	True	0.0	43.8	0.04						
5	False	45.08		11.08						
6	False	123.85	80.71	40.8						
7	True	0.0	52.55	0.04						
8	False	97.5		21.99						
9	False	249.93	100.36	63.08						

Figure 4.9 – Experimental results for multiprocess schedulings.

Chapter 5

Funnel automata: a heuristic approach

In this chapter we present a planning problem that can be solved using timed automata, as shown in [19]. This problem consists of building a controller for an automated system by finding a path in a timed automaton. As said before, the CEGAR approach that we used will not perform better than Uppaal in this example, since we know that our goal is reachable and we just want a path to this goal. We will present an example of such a case, and methods to speed up the process in that case.

5.1 Funnel automata

5.1.1 Control funnel

Let us consider a dynamical system, that is a system whose state is time dependent, and defined by its state $\boldsymbol{x} \in \mathbb{R}^d$ (with d the dimension of the system). For example $\boldsymbol{x}$ can represent the position or the velocity of an object. We use the notation $\dot{\boldsymbol{x}} = \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t}$ with $t \in \mathbb{R}_{\geq 0}$ representing time. We assume that this time is measured by a controller such that it is continuous and never reset. Let us assume that this system is controlled by a law

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, u(\boldsymbol{x}, t)) \tag{5.1}$$

In this equation we have a function $u : \mathbb{R}^d \times \mathbb{R}_{\geq 0} \to \mathbb{R}^k$ called the *control input function* that represents control laws on the systems, and $f : \mathbb{R}^d \times \mathbb{R}^k \to \mathbb{R}^d$ that is a continuous differentiable function that ensures the uniqueness of the solution for an initial state.

For a dynamical system such as this one, a *control funnel* is a function $\mathcal{F} : I \to 2^{\mathbb{R}^d}$. where I is an interval of $\mathbb{R}_{\geq 0}$ such that

$$\forall t \in I, \boldsymbol{x}(t) \in \mathcal{F}(t) \Rightarrow \forall t' \in I \text{ s.t. } t < t', \boldsymbol{x}(t') \in \mathcal{F}(t') \tag{5.2}$$

This means that if at a time $t \in I$, a trajectory is in the area defined by the funnel $\mathcal{F}(t)$, then it stays in that funnel in the future bounded by I. In other words, the control funnel does not define a trajectory but a constraint in which the trajectory has to stay. These funnels were introduced in [19] and are inspired from the notion of viability tubes that can be found in [10].

Example: dampened pendulum. For example let us assume a simple model of a pendulum as shown in Figure 5.1 and let us consider friction that creates dampening. If we consider θ the angle of the pendulum, a model for such a pendulum is driven by the differential equation $\ddot{\theta} = -A\theta - B\sin(C\theta)$ with A, B, C constants dependent of the physical properties of our system. This corresponds to what we previously defined with $\boldsymbol{x} = (\theta, \dot{\theta})$ our states constrained by this equation. In a perfect model with no dampening, we have a pendulum oscillating between the angles θ_0 and $-\theta_0$ with a maximum angular speed of $\sqrt{A}\theta_0$. This means that if we represent the trajectory of $\boldsymbol{x}$ it would be an ellipse (or a circle if the scale is chosen appropriately). In the dampened case, our trajectory stays inside this ellipse, since the amplitude and the maximum angular momentum decrease over time. Since the pendulum is dampened, we can say that after some time t', the pendulum amplitude decreases and the trajectory $\boldsymbol{x}(t)$ for $t > t'$ will stay inside a smaller ellipse defined by this new amplitude. This allows us to define a funnel for this movement, as represented in Figure 5.2. Since we know that the amplitude is decreasing we know that that the trajectory will be contained in smaller ellipses as time grows.

Example controlled system Another example of funnels would be a trajectory tracking controller. Let us assume a reference trajectory $\boldsymbol{x}_{ref}$ that we want to follow. Of course we cannot always ensure $\boldsymbol{x}(t) = \boldsymbol{x}_{ref}(t)$ for every time step t, but we want to stay as close as possible to this reference trajectory. We can design a control law u that ensures that we do converge toward this reference trajectory. Then we would have a control funnel $\mathcal{F}$ that ensures that as long as we stay close the trajectory $\boldsymbol{x}_{ref}$ at a time t', meaning $\boldsymbol{x}(t') \in \mathcal{F}(t')$, then for all $t > t'$ by following the control law u, we ensure that we stay closer to $\boldsymbol{x}_{ref}$ by $\boldsymbol{x}(t) \in \mathcal{F}(t)$ (of course we have $\boldsymbol{x}_{ref}(t) \in \mathcal{F}(t)$ for all t). Such an example can be seen in Figure 5.3. On this figure you can see the funnel converging to the reference trajectory, constraining the possible trajectories. Note that although the funnel tends to converge and diminish in size over time, this is not necessarly the case

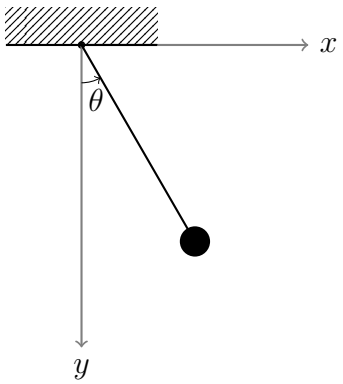


Figure 5.1 – A simple example of a pendulum

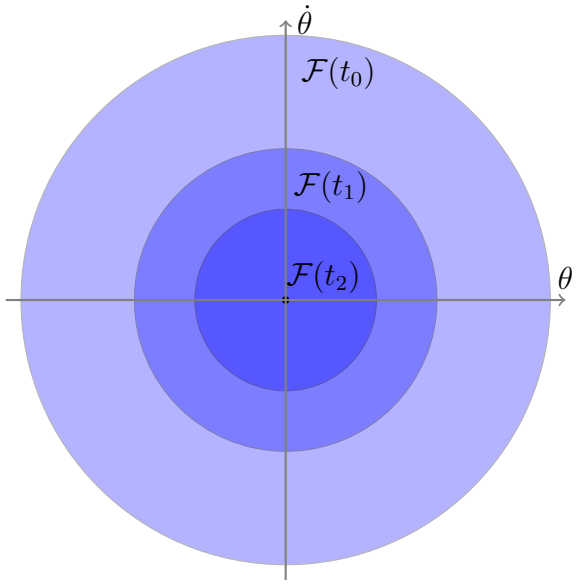


Figure 5.2 – An example of a funnel for the dampened pendulum, here with $t_0 < t_1 < t_2$

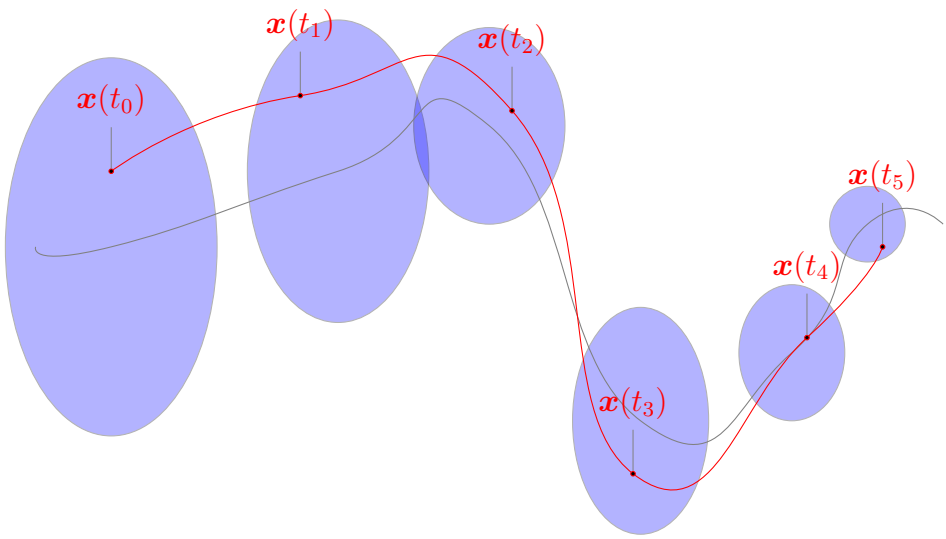


Figure 5.3 – An example of a funnel for a trajectory tracking controller. The dotted line represents the reference trajectory, and the red line represents a possible trajectory. We represent the funnel at different times $t_0 < t_1 < t_2 < t_3 < t_4 < t_5$

5.1.2 Motion planning with funnels

Let us assume a set of control laws $u_1(\boldsymbol{x}, t), u_2(\boldsymbol{x}, t), ..., u_n(\boldsymbol{x}, t)$. These can be seen as different reference trajectories for different movements for example. In a state $\tilde{\boldsymbol{x}}$ we can switch to a control law u_i if and only if there is a trajectory $\boldsymbol{x}(t)$ solution of $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, u_i(\boldsymbol{x}, t))$ and a time $t_0 \in \mathbb{R}_{\geq 0}$ such that $\tilde{\boldsymbol{x}} = \boldsymbol{x}(t_0)$. In other words, if we are in a state such that we can follow a trajectory that is a solution for u_i at a time t_0, then we can switch to the control law u_i. Note that here, we do not add a constraint of time in order to switch to the control law u_i. This means that before the switch occurred, we might be following another control law with a trajectory $\boldsymbol{x}'$ and $\boldsymbol{x}'(t_1) = \tilde{\boldsymbol{x}} = \boldsymbol{x}(t_0)$ with $t_1 \neq t_0$ meaning that when we switch the control law, we switch our referenced time to the time of this new trajectory. Let us define the *motion planning problem*

Definition 5.1.1. *For an initial state $\boldsymbol{x}_{init}$, a target $T_f \subseteq \mathbb{R}^d$ and an obstacle $\Omega \subseteq \mathbb{R}^d$, for a set of control law $u_1, u_2, ..., u_n$, the motion planning problem is to answer whether or not there are a sequence of indices $k_0, k_1, ..., k_N$ and two sequences of time $t_0^{start}, t_1^{start}, ...t_N^{start}$ and $t_0^{end}, t_1^{end}, ...t_N^{end}$ such that*

- *for $0 \leq i \leq N$, $t_i^{start} \leq t_i^{end}$*
- *for $0 \leq i \leq N$ there is a unique solution to $\dot{\boldsymbol{x}} = f(\boldsymbol{x}, u_{k_i}(\boldsymbol{x}, t))$ denoted $\boldsymbol{x}_i$ with $\boldsymbol{x}_i(t_i^{start}) = \boldsymbol{x}_{i-1}(t_{i-1}^{end})$ (except for $i = 0$ as $\boldsymbol{x}_0(t_0^{start}) = \boldsymbol{x}_{init}$)*
- *for $0 \leq i \leq N$ and a time t such that $t_i^{start} \leq t \leq t_i^{end}$), $\boldsymbol{x}_i(t) \notin \Omega$*
- *$\boldsymbol{x}_N(t_N^{end}) \in T_f$*

In other words, this means that there is a sequence of control laws that we can follow in succession, in order to reach T_f from $\boldsymbol{x}_{init}$ while avoiding the obstacle Ω. The previous definition expresses that we can effectively switch from one control law to the next one.

This problem can be modeled with funnels. Indeed, let us assume that for each control law u_i we have a sequence of funnels $\mathcal{F}_i^1, \mathcal{F}_i^2, ..., \mathcal{F}_i^{n_i}$. We denote by I_i^j the interval on which the funnel $\mathcal{F}_i^j$ is defined, and we do not consider funnels such that $\exists t \in I_i^j, \mathcal{F}_i^j(t) \cap \Omega \neq \emptyset$. By the definition of a funnel, we know that if for some value $t_0 \in I_i^j$, the current state is in $\mathcal{F}_i^j(t_0)$, then as long as we follow the control law u_i, for any $t > t_0$ with $t \in I_i^j$, the state of the system will remain $\mathcal{F}_i^j(t)$. This means that if there is $t_1 \in I_i^j$ and $t_2 \in I_p^q$, with $t_1 > t_0$ and $\mathcal{F}_i^j(t_1) \subseteq \mathcal{F}_p^q(t_2)$ then we know that as long as we reach $\mathcal{F}_i^j(t_0)$, then by following the control law u_i we know that we will be able to switch to the control law u_p in the future. So there is a way to switch from a funnel to another funnel and if this

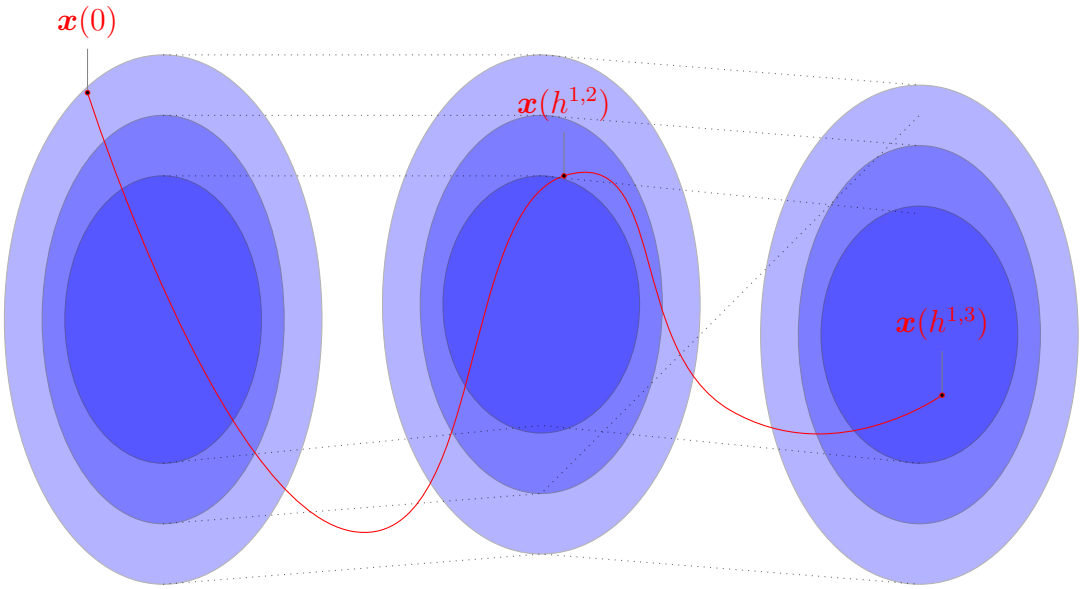


Figure 5.4 – An example of absorbing funnels. The red line represents a possible trajectory. We represent the funnels for the same control law $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$. Due to the absorbing relation, the trajectory starts in $\mathcal{F}_1$ but is the successively confined to $\mathcal{F}_2$ and $\mathcal{F}_3$

switch is possible, so is the switch from the represented control laws. Note that this is only an implication as a switch might be possible between two control laws, but not be possible between two funnels. So the funnel can be seen as an abstraction of the system, where the set of possible paths is under approximated.

In [19], it is also shown a different type of transition from a funnel to another called an *absorption*. Indeed, for any law u the case we study here follows the dynamics of the form $\dot{\boldsymbol{x}} = \dot{\boldsymbol{x}}_{ref} A(\boldsymbol{x} - \boldsymbol{x}_{ref})$ following a reference trajectory, where A is a matrix $\mathbb{R}^d \times \mathbb{R}^d$ whose eigen values have a negative real part. From this it can be deduced that we can find a sequence of funnels $\mathcal{F}^1, ...\mathcal{F}^k$ following the control law u such that for all i, j with $i < j$, we have $\forall t, \mathcal{F}_j(t) \subset \mathcal{F}_i(t)$ there is a constant $h^{i,j}$ such that for any trajectory $\boldsymbol{x}$ following u, if $\boldsymbol{x}(t) \in \mathcal{F}_i$ then $\boldsymbol{x}(t + h^{i,j}) \in \mathcal{F}_j$. In other words, we have a sequence of funnels, such that as we progress in this sequence, the funnels get tighter around the reference trajectory, and after staying in a funnel for a certain time, we know that we are in fact in a tighter funnel. Such funnels are represented in Figure 5.4

w

With this absorption, if we have a set of funnels $\mathcal{F}_i^j$, we can define transitions from a funnel $\mathcal{F}_i^j$ to $\mathcal{F}_p^q$ by either switching, or by absorption if $i = p$ and the absorption is possible, with a constant $h_i^{j,q}$. Now that we have a full abstraction of our model, the

problem becomes easier to solve.

Let us define a set of clocks $\mathcal{C} = \{c_h, c_t, c_g\}$ where c_g is a global clock that will never be reset, c_h is a clock measuring how long we have spent in the current funnel, and c_t is a clock used to track where we are in the current funnel. If we consider the pairs $(\mathcal{F}_i^j, v)$ with v a clock valuation, we can go from $(\mathcal{F}_i^j, v)$ to $(\mathcal{F}_p^q, v')$ in three different ways:

— With a *delay transition* if $p = i$, $q = j$ and there is a value $d \in \mathbb{R}_{\geq 0}$ such that $v' = v + d$, with $[v(c_t), v'(c_t)] \subseteq I_i^j$.

— With a *switching transition* if $v(c_g) = v'(c_g)$, $v'(c_h) = 0$, $v(c_t) \in I_i^j$, $v'(c_t) \in I_p^q$ and $\mathcal{F}_i^j(v(c_t)) \subseteq \mathcal{F}_p^q(v'(c_t))$.

— With an *absorption transition* if $p = i$, with an absorption from $\mathcal{F}_i^j$ to $\mathcal{F}_i^q$ $v(c_g) = v'(c_g)$, $v'(c_h) = 0$, $v(c_t) \in I_i^j$, $v'(c_t) = v(c_t)$ and $v(c_h) \geq h_i^{j,q}$.

From this transition system, we can easily show that if we can take a sequence of transitions from $(\mathcal{F}_0, v_0)$ to $(\mathcal{F}_f, v_f)$ with $\boldsymbol{x}_{init} \in \mathcal{F}_0(v_0(c_t))$ and $\mathcal{F}_f(v_f(c_t)) \cap T_f \neq \emptyset$, then we can plan a motion. Moreover, by looking at the control law associated to each funnel, and the time at which the switch is done, we can deduce an effective planning for the motion. Note that this is only an implication and not an equivalence, as going from motion to funnel made us under approximate the possible paths. We remark that we can have switching transitions between two funnels of the same reference trajectory x_{ref}^i, from $\mathcal{F}_i^j$ to $\mathcal{F}_i^q$, even though there is an absorption transition from $\mathcal{F}_i^q$ to $\mathcal{F}_i^j$. This is because while both funnels describe the same trajectory, switching from one to the other allow to shift the value of c_t from t to t', allowing us to follow a point of the trajectory $x_{ref}^i(t')$ that is slightly behind or slightly ahead of the point $x_{ref}^i(t)$ that we were following before, as shown in Figure 5.5.

5.1.3 Funnel automata

From there we can reduce this problem to an reachability problem in timed automata, as shown in [19]. This requires to consider an extension of the previously defined timed automata, where instead of resetting a clock to 0, we can set it to any value. This is not a problem since the number of values that a clock can be set to is limited. Indeed we can just see it as a clock reset where we remember the offset, since the number of offsets is limited, and this can be encoded in the location. So we will note the reset R as a partial function on the set of clocks that assigns clocks to their new value. We can build this timed automaton $(\Sigma, \mathcal{L}, \mathsf{Inv}, \ell_0, \mathcal{C}, E)$ where :

— Σ is empty, since we are not considering synchronization here.

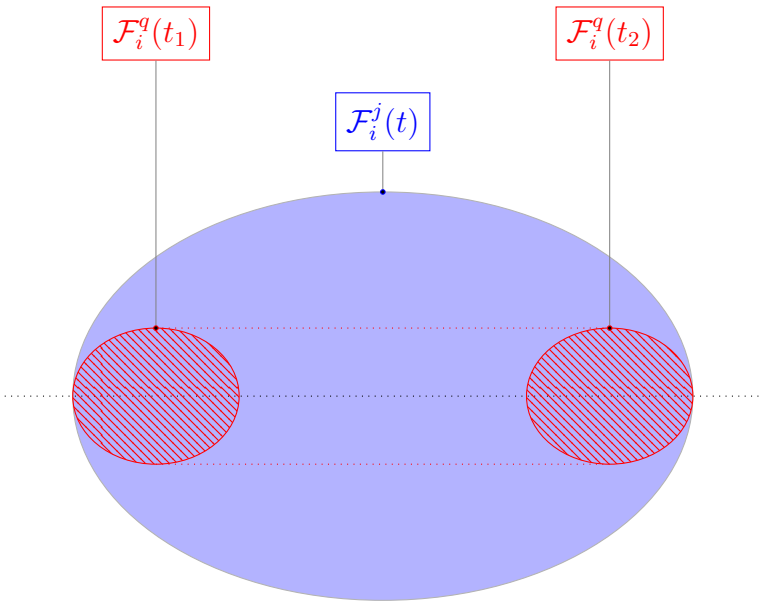


Figure 5.5 – For the same reference trajectory, we have two funnels $\mathcal{F}_i^j$ and $\mathcal{F}_i^q$, with an absorption from j to q. But here we also represent a switch from $\mathcal{F}_i^q$ to $\mathcal{F}_i^j$, allowing to change the value of c_t from any value between t_1 and t_2 to the value t.

— $\mathcal{L} = \{\mathcal{F}_i^j \mid 1 \leq i \leq n \text{ and } 1 \geq j \geq n_i\} \cup \{\mathsf{init}\}$

— $\mathsf{Inv}(\mathcal{F}_i^j) = c_t \in I_i^j$ that can be expressed as a finite number of comparison, under approximating I_i^j if necessary.

— $\mathcal{C} = \{c_h, c_t, c_g\}$

— E is defined as $E_{init} \cup E_{switch} \cup E_{absorb}$ with

1. $E_{init} = \{(\mathsf{init}, true, R, \mathcal{F}_i^j) \mid \exists t \in I_i^j; \boldsymbol{x}_{init} \in \mathcal{F}_i^j(t) \wedge R(c_g) = R(c_h) = 0 \wedge R(c_t) = t\}$
2. $E_{switch} = \{(\mathcal{F}_i^j, \alpha \leq c_t \leq \beta, R, \mathcal{F}_p^q) \mid [\alpha, \beta] \subseteq I_i^j \wedge \exists t' \in I_p^q, R(c_h) = 0 \wedge R(c_t) = t' \wedge \forall t \in [\alpha, \beta], \mathcal{F}_i^j(t) \subseteq \mathcal{F}_p^q(t')\}$
3. $E_{absorb} = \{(\mathcal{F}_i^j, \alpha \leq c_h, R, \mathcal{F}_i^q) \mid \mathcal{F}_i^j \text{ is absorbed by } \mathcal{F}_i^q \text{ with } h_i^{j,q} = \alpha \wedge R(c_h) = 0\}$

The main problem with this timed automaton is that it is not finite. Indeed, the number of edges can be infinite, since there can be an infinity of values α, β, t' for which $\forall t \in [\alpha, \beta]\mathcal{F}_i^j(t) \subseteq \mathcal{F}_p^q(t')$. In order to restrict ourselves to a finite model, we will choose a finite subset of E so that we only have a finite number of transitions. From this construction, we can show that if we can reach a location $\mathcal{F}_i^j$ at a valuation v such that $\mathcal{F}_i^j(v(c_t)) \in T_f$, then there is a sequence of transition in the transition system defined in the previous

section, from $(\mathcal{F}_0, v_0)$ to $(\mathcal{F}_f, v_f)$ with $\boldsymbol{x}_{init} \in \mathcal{F}_0(v_0(c_t))$ and $\mathcal{F}_f(v_f(c_t)) \cap T_f \neq \emptyset$. Note that we have once again underestimated the number of possible paths, since we limited ourselves to a finite number of edges, and that is why we only have an implication. But if we find such a path in the timed automaton, we can deduce a planning for the motion.

5.1.4 Pick and place problem

We will consider a simple system, of a grabber that can move on a rail. Perpendicular to this rail are 3 lanes that bring objects or packages to place in a goal. We number these lanes from 1 to 3 and we consider a fourth lane called lane 0 where we want to drop the objects. We consider that the lanes are equidistant and we will only monitor the position x of the grabber along the rail and its speed $\dot{x}$. This two values are the state that we monitor $\boldsymbol{x}$. The goal here is to find a controller so that the grabber can pick up all the packages as they arrive, and drop them on the lane 0. This problem, and its resolution using timed automata was introduced in [19]. We denote the position of a lane i by x_{L_i}. Due to the limitations of the system, the speed and acceleration of the grabber are constrained. We will consider a certain number of positive reference velocities, noted N_{vel} and a reference trajectory for each of these velocities denoted by $\boldsymbol{x}^i_{ref}$ with $0 < i \leq N_{vel}$, ordered from the slowest to the fastest. Each trajectory represents a movement from x_{L_0} to x_{L_4} at a constant speed. For each of these trajectory $\boldsymbol{x}^i_{ref}$ we consider the trajectory $\boldsymbol{x}^{-i}_{ref}$ the trajectory from x_{L_4} to x_{L_0} with an opposite velocity. Moreover we define the static trajectories $\boldsymbol{x}^{L_k}_{ref}$ for $k \in \{0, ..., 3\}$ that represent a stationary position in x_{L_k}.

We also associate each trajectory x^i_{ref} to a controller that defines a control law u^i_{ref}. This control low ensures that we can define a sequence of funnels $\mathcal{F}^0_i, ..., \mathcal{F}^{N_{fun}}_i$ such that for all $j, j' \in \{0, ..., N_{fun}\}$, $\mathcal{F}^j_i$ being a funnel of constant size, $I^j_i = I^{j'}_i$, $\mathcal{F}^j_i$ is absorbed in $\mathcal{F}^{j+1}_i$ and if $j \leq j'$, then $\forall t \in I^j_i, \mathcal{F}^{j'}_i(t) \subseteq \mathcal{F}^j_i(t)$. Note that we then have a set of funnels that depends on the number of reference velocities N_{vel} and the number of funnels defined for each of these velocities N_{fun}. Moreover, the set of possible switches between two control laws is dependent on the definition of these funnels. In order to allow the switch between different trajectories, we ensure that for two neighboring trajectories $\boldsymbol{x}^i_{ref}$ and $\boldsymbol{x}^k_{ref}$, meaning that they either have a neighboring index or one of them is stationary while the other is of the slowest velocity, then we have $\forall t \in I^0_i, \exists t' \in I^{N_{fun}}_k, \mathcal{F}^0_i(t) \subseteq \mathcal{F}^{N_{fun}}_k(t')$. In other words, this means that if we know that our current trajectory is close to the reference trajectory we are following, so we know that our velocity is close to the reference one, then we can switch to the closest trajectory but we are know farther away from our

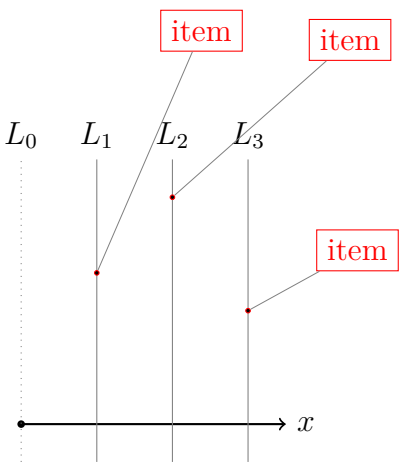


Figure 5.6 – The pick and place problem, with the 3 lanes, and 1 item in each lane

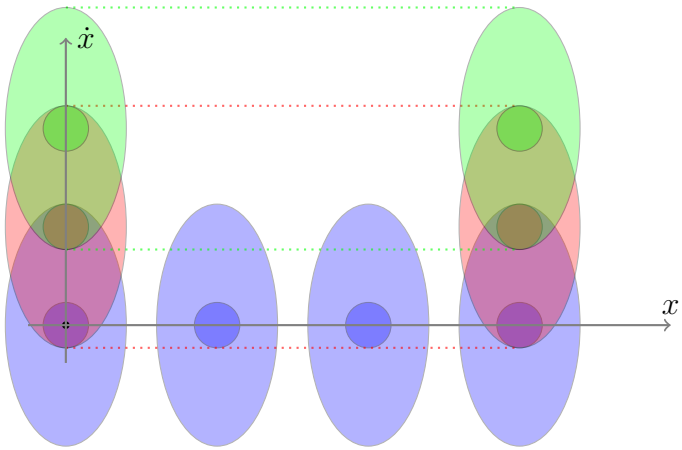


Figure 5.7 – A representation of the funnels for pick and place, with 2 reference velocities and 2 funnels per velocity. The green funnels represent the maximum velocity and the red ones represent a lower velocity. The blue funnels represent the stationary trajectories for each of the 4 lanes.

new reference trajectory. We represent the problem and the funnel generated in Figure 5.7

We can build a timed automaton as we described before. We consider that, at each time step t, the funnel is defined as an ellipse so intersections can be computed easily. We compute an arbitrary amount of switching transition, by looking for a finite number of these intersections. Moreover, we need to make a small timed automaton that models the other constraints on the system, such that the timing at which the packages arrive, modeled by a constraint on the global clock. Also the fact that only 2 packages can be held at the same time, that we need to deposit all the packages (in this case 3, one for each lane) in the lane 0, and that we need to be on the right position at a slow velocity to grab a package. This is done with a small 2 state automaton, that we call the *task automaton*, and by doing the product between these two timed automata, we can solve our problem. Note that the funnel automata is an abstraction of our problem that underestimates the number of paths. By increasing the number of velocities or funnels, we increase the number of states, and by increasing the number of possible intersections, we increase the number of switching transitions. If these numbers are not high enough, the target state might not be reachable. If we find that the goal state is reachable, we then have a path giving us all the control laws and the switch that we need to take to complete the task.

5.2 Algorithm for a pick-and-place problems

As we have shown, we can build an automaton, and by applying a reachability algorithm we can deduce a planning, so this problem could be solved using Uppaal on the resulting timed automata, but we propose an improvement. The main problem here is that the algorithms defined in the previous chapters are not adapted to this situation. Indeed, here we want to prove that a "good" state is reachable instead of proving that we can never reach a "bad" state. This means that algorithms that over-approximate the number of paths will not help us, as we are only looking for real paths here. So we need algorithms that under approximates the number of paths, since that is what is used to build the timed automaton. We can notice that underestimating the number of paths, before, and adding paths that we omitted if we cannot reach the goal, is the same as exploring our automata in a certain order. So it is equivalent to a heuristic search.

5.2.1 A heuristic approach

So we chose a heuristic approach to solve this problem, and that is to prioritize the search in the zone graph. Indeed, since we are only interested in finding a single path, exploring the entirety of the graph is unnecessary work. So we define a function h that associates each node n to a priority, that is a positive integer. Then when we pick an element n from wait we pick the one with the lowest priority $h(n)$ first. We consider the same algorithm as the one regularly used to solve reachability, the only difference being the implementation of wait, that is now a priority queue parametrized by a function h.

Naive notion of predictive distance. A first idea is to associate to each location a distance to the target location. It means that $h(n)$ is the minimal number of transitions between $n.\ell$ and the goal, without taking the clocks into account. So that way we ignore the locations that cannot reach the goal, and we try to find a path that switches between funnels the least amount of times. While this approach might be good, probably for simpler funnel examples, it is not efficient in this case since the task is very timing dependent. Indeed our packages arrive at very precise timings, and the paths with the least amount of switches or absorptions are paths that stay at the lowest possible velocity and often arrive too late. Moreover, computing this notion of distance requires some preprocessing. We can still say that if the traditional algorithms that use depth-first search find a target location at a depth k, then this means that the target can be reached by taking $k - 1$ transitions.

So in theory, with this heuristic, we should only compute paths of length less than k before finding that the target is reachable, and we can hope that we will explore less than the depth-first search algorithm. Unfortunately, our tests have proven that this is not the case and that this heuristic is slightly slower if the goal is reachable, and noticeably slower when it is not reachable. We assume that this due to the preprocessing, but also that this algorithm starts with deep explorations of paths that have no chance to reach the goal location, because the movement is too slow, leading up to high values for the clocks c_g being explored first, while they would have been covered by the relation of subsumption by using depth-first search.

Measuring the elapsed time. The second idea is to measure the elapsed time, and prioritize the paths that reach our goals quickly. This can be done easily by defining $h(n) = h_{elapsed}(n.Z)$ where $h_{elapsed}(Z)$ is the lower bound on c_g in Z. The intuition is that this will help us find the paths that stop in a lane as soon as possible, even if we have to wait in this lane. This will also solve the covering problems, since we will explore the lowest values of c_g first. Also, this does not require any preprocessing. Finding the path that reach the goal in a minimal time is not a new problem and can be found for example in [9] referred as *minimal-time reachability*. The problem with this method is that there are a lot of switching and absorption possible before picking up the first package, and some of them are of no use for us, e.g. slowing down and accelerating in succession tends to create a long path that we do not necessary want to explore.

Measuring the past while estimating the future: the A* algorithm. Our solution is to combine both methods cited previously into something that is inspired by the A* algorithm. Indeed the A* algorithm tries to find the shortest path by measuring the distance up to the location, and estimating a distance to the goal. First we define an estimate function h_{est} that associates to each edge an estimated time to reach the goal. We first define a function f_{est} that associates to a pair $(e, \ell) \in E \times \mathcal{L}$ the underestimated time to reach ℓ by taking e. For a location ℓ, let us denote by $absorb(\ell)$ the set of locations reachable from ℓ by taking only absorption transitions, ℓ included. Let us denote $switch(\ell)$ the set of switch transitions starting in ℓ and $switch^*(\ell) = \bigcup_{\ell' \in absorb(\ell)} switch(\ell)$. This means that from ℓ, the next reset that can occur on c_t is from a transition of $switch^*(\ell)$. For a switch transition e, we denote α_t^e, β_t^e and r_t^e the lower bound on c_t, the upper bound on c_t and the value to which c_t is set respectively. We note $delay(k, \alpha, \beta)$ the function

that returns 0 if $k \in [\alpha, \beta]$, $\alpha - k$ if $k \leq \alpha$ and $+\infty$ otherwise. We ensure that for every location ℓ and for every transition $e = (\ell_1, g, a, R, \ell_2)$:

— if $\ell_2 = \ell$ then $f_{est}(e, \ell) = 0$

— if e is a switch transition, then $f_{est}(e, \ell) = \min_{e' \in switch^*(\ell_2)}(delay(r_t^e, \alpha_t^{e'}, \beta_t^{e'}) + f_{est}(e', \ell))$

— if e is an absorption transition then $f_{est}(e, \ell) = \min_{e' \in switch^*(\ell)} f_{est}(e', \ell))$

This means that for a switching transition e, we estimate the minimum time that can be spent before taking another switching transition e' by comparing the value of c_t set by e, and the constraint on c_t in e'. If e is an absorption transition, then we just consider the next reachable switching transitions. Of course this is a big under approximation since we cannot know the value of c_t simply by taking the transition e, and thus we consider that we can take every transition from there. From this, for a target location ℓ_f, we can set $h_{est}(e) = f_{est}(e, \ell_f)$. This function can be computed recursively by first computing the value for every edge going to ℓ_f, using Section 5.2.1 and setting the value to $+\infty$ for every other edge. Then we can iterate Section 5.2.1 on every edge e pointing to a location ℓ such that an edge of $switch^*(\ell)$ had its value modified during previous iteration, since the second property only takes switching transitions into account. Finally, once the values have been computed for switching transition we can also compute them for the absorption transitions.

Now we can define $h(n) = h_{elapsed}(n.Z) + h_{est}(n.parent)$. where $n.parent$ is the edge used to create the node n. We have a function $h_{elapsed}$ measuring the effective time needed to reach the node n and a function h_{est} that estimates what time is needed to reach the target ℓ_f.

5.2.2 Results

We implemented this heuristic in TChecker and tested it on several examples of funnel automata for the pick and place problem. We generated these automata using a python script, the difference between these timed automata being the number of reference velocities, the number of funnels for a velocity, and the number of intersections looked at for defining the switch constraints. We also noted the time for the preprocessing, that is the computation of h_{est}. We run those tests on the same machine as the one used in the previous chapter, with a timeout of 1 hour, and present the results in Table 5.1

Model	Reachable	Uppal time(sec)	Heuristic time(sec)	preprocess (sec)
`funnel_2_3_2`	true	847	851	234
`funnel_4_5_2`	true	1341	1283	394
`funnel_5_6_2`	true	2896	2545	521
`funnel_3_4_4`	true	3051	timeout	987
`funnel_4_5_1`	false	1292	1682	171
`funnel_2_3_1`	false	645	1088	87

Table 5.1 – Comparative results between Uppaal and our heuristic approach. The name of a file indicates the number of reference velocities, the number of subfunnels, and a factor indicating the number of switches created. For each model, we say whether or not the timed automaton has indeed a path that delivers all 3 packages, as well as the preprocessing time necessary for the heuristic.

The first thing that can be noted here is that this heuristic is not working well if the goal is not reachable, that is if we have a small number of funnels and switches between them. This is expected since we need to explore the entirety of the zone graph in that case, and we are losing time in preprocessing. Note that in that case, the time lost in preprocessing is not the only factor. We suppose that the other problem is that the covering relation generated by this order of exploration is less optimal than the one generated by a depth-first search. That being said, this is not the case that interests us, since this heuristic was not designed to be efficient in the non reachable case. On the reachable case, the results are a bit better, although the preprocessing seems to be growing with the number of switches generated, which is to be expected. Interestingly, cases with a high number of switches seem also to be the ones where the gain is high, especially discounting the preprocessing. We also recognize that the implementation for the computation of h_{est} is not optimal, and that optimizations can certainly be made. Moreover, the value computed by h_{est} can be discussed, especially for an absorption transition. Indeed while a switching transition is given a decent estimation, this is not the case for an absorption transition. Indeed the value for an absorption transition might be lower than the value for the switching transition that preceded it. This happens because we forget the value of c_t in the estimation h_{est} for an absorption transition, resulting in us relying on the values of switching transition that we might not be able to take. This could be improved by computing the values of h_{est} for absorption transitions during the exploration, by using the true value of c_t. This is not something we have done, especially since the impact of this is limited due to the fact that we only consider a small number of funnels per reference

trajectory, meaning that the number of consecutive absorptions is quite low.

Conclusion

Here we summarize the results presented in this thesis, and we also discuss possible improvements to our algorithms. We also discuss more general perspectives for this work.

Quantization abstraction

We have presented and defined the quantization abstraction for zones, and we have presented its implementation in two algorithms. The results presented here show multiple examples where this abstraction allows for faster verification of safety properties in timed automata.

Algorithm choice. We have shown that these algorithms are more efficient than Uppaal or PAT on some examples. We will note that these examples were proposed by us, and that our algorithms are less efficient on the more traditional models. We think that one of the reasons is that these algorithms are better suited to timed automata that are generated automatically, with some redundancy or states and constraints that are not relevant to the properties we are checking. On traditional models such as Fischer or CSMA/CD, we believe that our algorithms are less efficient because those models are “well designed", meaning that those models are designed in a way to minimize the state-space to explore, so that Uppaal would be efficient on these examples. As such, we think that the choice between our algorithms and the traditional non CEGAR algorithms can be decided by looking at the timed automaton and how it was obtained. We believe that on models generated automatically by other algorithms that may add superfluous states and constraints, the choice of our algorithms can be justified.

Domain choice. We also notice that the biggest difference in the results was not between the approach taken for the algorithms, but between the implementation of the abstract domain for the quantization abstraction. Indeed, for a global domain, the enumerative algorithm and the symbolic algorithm have comparable performances, and the noticeable difference is between the use of global domain versus the use of a localized domain. The

choice of the type of domain has to be motivated by the model on which the property is checked. Here there is less intuition. Global domains work better on models where only a few constraints are needed by the entirety of the model to prove that the goal is not reachable, while local domains are a better choice when the model needs a lot of constraints but the number of constraints needed per location is low. Of course this is something that might be harder to determine by looking at the model before running the algorithms.

Interpolant choice. We provided and detailed an algorithm to find an interpolant between two zones. This is a key part of our CEGAR loop, as this choice of interpolant is the refinement part of our loop. Our algorithm is designed to find the interpolant that involves the minimum number of constraints, in order to have the coarse abstraction possible. But the coarser abstraction is not always the best choice. This is evident since the global domain sometimes performs better than the localized domain despite the fact that the localized domain provides a coarser abstraction. So choosing the interpolant with the minimal number of constraints might not be the best option, and choosing an interpolant based on other things, like the constraints on the outgoing transitions might be better and worth implementing.

Predecessor computation. We also note that finding a refinement forces us to recompute some part of our exploration. Because of this, being able to detect a spurious trace earlier can be useful, allowing us to explore less before refinement. This can be achieved by using the predecessor computation that we do when we have a refinement. Using the enumerative algorithm for example, once a predecessor is computed, we know that if we reach the corresponding location, with an abstracted zone that intersects this predecessor, then by definition of the predecessors we can also reach the target state from this abstracted zone. It is the same as adding the pair location/predecessor to the set of target states, and as such would allow us to detect the refinements earlier. This should be more efficient, especially since our exploration is a breadth-first search. Of course this would require to check many intersections every time a new node is computed which might be too big of a cost, but we believe that this might be a worthwhile improvement.

Scalable symbolic algorithm. We will remark that while the symbolic algorithm is still a BDD based algorithm that is not dependent of the maximum constant that appear in the timed automaton. Unlike other BDD based algorithms, if all the constants of the timed automaton are scaled, the execution time of our algorithm does not increase.

Extensions. Finally we note that both approaches could be generalized to weighted timed automata or other extension of timed automata. We believe that the enumerative CEGAR algorithm could be adapted to weighted timed automata as algorithms on weighted timed automata use priced zone, that is zones that are associated with an affine function that determines the cost. If we only over approximates the zone in a direction that only increases cost, we do not add a new minima to the cost function. That is why we believe that be only considering over-approximations in directions that increase the cost, our algorithms could be applied as such, and this change could be easily tried as soon as tools for weighted timed automata are added to TChecker.

Heuristic approach

We have given an example of an heuristic approach on one case of funnel automata. This approach could be adapted for other use of these automata. Moreover we also note that while our experiments have shown that our implementation is barely faster than Uppaal in some cases and considerably slower in others, we have also shown that a lot of this lost time can be found in the computation of our heuristic, during the preprocessing. We believe that this preprocessing could be implemented in a better way, or that a simpler version of the heuristic could be used, allowing our heuristic approach to be applied more efficiently.

Moreover we believe that this heuristic approach could be combined with our CEGAR algorithms. Indeed a main problem of the CEGAR algorithms is the breadth-first search approach. While a depth-first search is not advisable and produces poor results, a compromise between the two approaches, based on an heuristic search could be an improvement for the enumerative CEGAR algorithm. Indeed, after a refinement, the nodes that have been refined, meaning the nodes from which we found a spurious trace, are more likely to produce another spurious trace in the future. Prioritizing them in the exploration might speed up the process. Moreover this could make the CEGAR approach more viable whenever the target state is reachable.

Bibliography

[1] Yasmina Abdeddaim and Oded Maler, « Job-Shop Scheduling Using Timed Automata? », *in*: vol. 2102, June 2001, DOI: 10.1007/3-540-44585-4_46.

[2] Ernst Althaus et al., « Verification of linear hybrid systems with large discrete state spaces using counterexample-guided abstraction refinement », *in*: *Science of Computer Programming* 148 (2017), Special issue on Automated Verification of Critical Systems (AVoCS 2015), pp. 123–160, ISSN: 0167-6423, DOI: https://doi.org/10.1016/j.scico.2017.04.010, URL: http://www.sciencedirect.com/science/article/pii/S0167642317300850.

[3] Rajeev Alur, Costas Courcoubetis, and David L. Dill, « Model-Checking in Dense Real-Time », *in*: *Information and Computation* 104.*1* (May 1993), pp. 2–34.

[4] Rajeev Alur and David L. Dill, « A Theory of Timed Automata », *in*: *Theoretical Computer Science* 126 (1994), pp. 183–235.

[5] Rajeev Alur and David L. Dill, « A Theory of Timed Automata », *in*: *Theoretical Computer Science* 126.*2* (1994), pp. 183–235.

[6] Rajeev Alur and David L. Dill, « Automata For Modeling Real-Time Systems », *in*: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings*, ed. by Mike Paterson, vol. 443, Lecture Notes in Computer Science, Springer, 1990, pp. 322–335, ISBN: 3-540-52826-1, DOI: 10.1007/BFb0032042, URL: https://doi.org/10.1007/BFb0032042.

[7] Étienne André and Jun Sun, « Parametric Timed Model Checking for Guaranteeing Timed Opacity », *in*: *Lecture Notes in Computer Science* (2019), pp. 115–130, ISSN: 1611-3349, DOI: 10.1007/978-3-030-31784-3_7, URL: http://dx.doi.org/10.1007/978-3-030-31784-3_7.

[8] Étienne André et al., « IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems », *in*: *FM 2012: Formal Methods*, ed. by Dimitra Giannakopoulou and Dominique Méry, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 33–36, ISBN: 978-3-642-32759-9.

[9] Étienne André et al., « Minimal-Time Synthesis for Parametric Timed Automata », *in*: *CoRR* abs/1902.03013 (2019), arXiv: 1902.03013, URL: http://arxiv.org/abs/1902.03013.

[10] Jean-Pierre Aubin, *Modelling and adaptive control : proceedings of the IIASA conference, Sopron, Hungary, July, 1986 / Viability Tubes*, eng, Lecture notes in control and information sciences, Springer-Verlag, ISBN: 3-540-19019-8.

[11] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, MIT Press, 2008, ISBN: 978-0-262-02649-9.

[12] Thomas Ball and Sriram Rajamani, « The SLAM toolkit », *in*: *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, ed. by Gérard Berry, Hubert Comon, and Alain Finkel, vol. 2102, Lecture Notes in Computer Science, Springer-Verlag, July 2001, pp. 260–264.

[13] Gerd Behrmann et al., « Lower and Upper Bounds in Zone Based Abstractions of Timed Automata », *in*: *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, ed. by Kurt Jensen and Andreas Podelski, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 312–326, ISBN: 978-3-540-24730-2, DOI: 10.1007/978-3-540-24730-2_25, URL: http://dx.doi.org/10.1007/978-3-540-24730-2_25.

[14] Gerd Behrmann et al., « Minimum-Cost Reachability for Priced Time Automata », *in*: *Hybrid Systems: Computation and Control*, ed. by Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 147–161, ISBN: 978-3-540-45351-2.

[15] Gerd Behrmann et al., « UPPAAL 4.0 », *in*: *Proceedings of the 3rd International Conference on Quantitative Evaluation of Systems (QEST'06)*, IEEE Comp. Soc. Press, Sept. 2006, pp. 125–126, DOI: 10.1109/QEST.2006.59.

[16] Johan Bengtsson and Wang Yi, « Timed Automata: Semantics, Algorithms and Tools », *in*: *Lectures on Concurrency and Petri Nets*, ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, vol. 2098, Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 87–124, DOI: 10.1007/b98282.

[17] Johan Bengtsson et al., « Uppaal in 1995 », *in*: *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS '96 Passau, Germany, March 27–29, 1996 Proceedings*, ed. by Tiziana Margaria and Bernhard Steffen, Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 431–434, ISBN: 978-3-540-49874-2, DOI: 10.1007/3-540-61042-1_66, URL: http://dx.doi.org/10.1007/3-540-61042-1_66.

[18] Bernard Berthomieu and Miguel Menasche, « An Enumerative Approach for Analyzing Time Petri Nets », *in*: *Information Processing 83 – Proceedings of the 9th IFIP World Computer Congress (WCC'83)*, ed. by R. E. A. Mason, North-Holland/IFIP, Sept. 1983, pp. 41–46.

[19] Patricia Bouyer et al., « Timed-automata abstraction of switched dynamical systems using control invariants », *in*: *Real-Time Systems* (2017), pp. 1–27, DOI: 10.1007/s11241-016-9262-3, URL: https://hal.sorbonne-universite.fr/hal-01436413.

[20] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled, *Model checking*, MIT Press, 2000.

[21] Edmund M. Clarke et al., « Counterexample-Guided Abstraction Refinement for Symbolic Model Checking », *in*: *Journal of the ACM* 50.*5* (Sept. 2003), pp. 752–794, DOI: 10.1145/876638.876643.

[22] Maximilien Colange, Dimitri Racordon, and Didier Buchs, « A CEGAR-like Approach for Cost LTL Bounds », *in*: *CoRR* abs/1506.05728 (2015), arXiv: 1506.05728, URL: http://arxiv.org/abs/1506.05728.

[23] Patrick Cousot and Radhia Cousot, « Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints », *in*: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, Los Angeles, California: ACM, 1977, pp. 238–252, DOI: 10.1145/512950.512973, URL: http://doi.acm.org/10.1145/512950.512973.

[24] David L. Dill, « Timing Assumptions and Verification of Finite-State Concurrent Systems », *in*: *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems (AVMFSS'89)*, ed. by Joseph Sifakis, vol. 407, Lecture Notes in Computer Science, Springer, 1990, pp. 197–212.

[25] Rüdiger Ehlers, Robert Mattmüller, and Hans-Jörg Peter, « Combining Symbolic Representations for Solving Timed Games », *in*: *Proceedings of the 8th International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'10)*, ed. by Krishnendu Chatterjee and Thomas A. Henzinger, vol. 6246, Lecture Notes in Computer Science, Springer-Verlag, Sept. 2010, pp. 107–212, DOI: 10.1007/978-3-642-15297-9_10.

[26] R. Ehlers et al., « Fully Symbolic Timed Model Checking Using Constraint Matrix Diagrams », *in*: *2010 31st IEEE Real-Time Systems Symposium*, Nov. 2010, pp. 360–371, DOI: 10.1109/RTSS.2010.36.

[27] Thomas A. Henzinger et al., « Lazy Abstraction », *in*: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, Portland, Oregon: ACM, 2002, pp. 58–70, ISBN: 1-58113-450-9, DOI: 10.1145/503272.503279, URL: http://doi.acm.org/10.1145/503272.503279.

[28] Thomas A. Henzinger et al., « Software verification with BLAST », *in*: *Proceedings of the 10th International SPIN Workshop (SPIN'03)*, ed. by Thomas Ball and Sriram Rajamani, vol. 2648, Lecture Notes in Computer Science, Springer-Verlag, Apr. 2003, pp. 235–239.

[29] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz, « Better Abstractions for Timed Automata », *in*: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, 2012, pp. 375–384, DOI: 10.1109/LICS.2012.48, URL: https://doi.org/10.1109/LICS.2012.48.

[30] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz, « Lazy abstractions for timed automata », *in*: *CoRR* abs/1301.3127 (2013), URL: http://arxiv.org/abs/1301.3127.

[31] Frédéric Herbreteau and Thanh-Tung Tran, « Improving Search Order for Reachability Testing in Timed Automata », *in*: *Proceedings of the 13th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'15)*, vol. 9268, Lecture Notes in Computer Science, Springer, 2015, pp. 124–139.

[32] Frédéric Herbreteau et al., « Using non-convex approximations for efficient analysis of timed automata », *in*: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, 2011, pp. 78–89, DOI: 10.4230/LIPIcs.FSTTCS.2011.78.

[33] Swen Jacobs et al., « The 5th Reactive Synthesis Competition (SYNTCOMP 2018): Benchmarks, Participants & Results », *in*: *CoRR* abs/1904.07736 (2019), arXiv: 1904.07736, URL: http://arxiv.org/abs/1904.07736.

[34] A. Jovanović, D. Lime, and O. H. Roux, « Integer Parameter Synthesis for Real-Time Systems », *in*: *IEEE Transactions on Software Engineering* 41.*5* (2015), pp. 445–461, DOI: 10.1109/TSE.2014.2357445.

[35] Kim G. Larsen, Paul Pettersson, and Wang Yi, « UPPAAL in a nutshell », *in*: *Int. Journal on Software Tools for Technology Transfer* 1 (1997), pp. 134–152.

[36] Kenneth L. McMillan, « Symbolic Model Checking — An Approach to the State Explosion Problem », PhD thesis, Pittsburgh, Pennsylvania, USA: Carnegie Mellon University, 1993.

[37] Takeshi Nagaoka, Kozo Okano, and Shinji Kusumoto, « An Abstraction Refinement Technique for Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop », *in*: *IEICE Transactions* 93-D.*5* (2010), pp. 994–1005, URL: http://search.ieice.org/bin/summary.php?id=e93-d_5_994.

[38] Truong Khanh Nguyen et al., « Improved BDD-Based Discrete Analysis of Timed Systems », *in*: *FM*, 2012, pp. 326–340.

[39] Truong Khanh Nguyen et al., « Scaling BDD-based Timed Verification with Simulation Reduction », *in*: *Formal Methods and Software Engineering*, 2016, pp. 363–382.

[40] Kozo Okano, Behzad Bordbar, and Takeshi Nagaoka, « Clock Number Reduction Abstraction on CEGAR Loop Approach to Timed Automaton », *in*: *Second International Conference on Networking and Computing, ICNC 2011, November 30 - December 2, 2011, Osaka, Japan*, IEEE Computer Society, 2011, pp. 235–241, ISBN: 978-0-7695-4569-1, DOI: 10.1109/ICNC.2011.42, URL: https://doi.org/10.1109/ICNC.2011.42.

[41] Victor Roussanaly, Ocan Sankur, and Nicolas Markey, « Abstraction Refinement Algorithms for Timed Automata », *in*: *CoRR* abs/1905.07365 (2019), arXiv: 1905.07365, URL: http://arxiv.org/abs/1905.07365.

[42] Sanjit A. Seshia and Randal E. Bryant, « Unbounded, Fully Symbolic Model Checking of Timed Automata Using Boolean Methods », *in*: *Computer Aided Verification*, ed. by Warren A. Hunt and Fabio Somenzi, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 154–166, ISBN: 978-3-540-45069-6.

[43] Jiří Srba, « Timed-Arc Petri Nets vs. Networks of Timed Automata », *in*: *Applications and Theory of Petri Nets 2005*, ed. by Gianfranco Ciardo and Philippe Darondeau, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 385–402, ISBN: 978-3-540-31559-9.

[44] Tamás Tóth and István Majzik, « Lazy reachability checking for timed automata using interpolants », *in*: *Proceedings of the 15th International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'17)*, ed. by Alessandro Abate and Gilles Geeraerts, vol. 10419, Lecture Notes in Computer Science, Springer-Verlag, Sept. 2017, pp. 264–280, DOI: 10.1007/978-3-319-65765-3_15.

[45] Stavros Tripakis and Sergio Yovine, « Analysis of Timed Systems Using Time-Abstracting Bisimulations. », *in*: *Formal Methods Syst. Des.* 18.*1* (2001), pp. 25–68, URL: http://dblp.uni-trier.de/db/journals/fmsd/fmsd18.html#TripakisY01.

[46] Farn Wang, « Symbolic Verification of Complex Real-time Systems with Clock-restriction Diagram », *in*: *Formal Techniques for Networked and Distributed Systems*, ed. by Myungchul Kim et al., Boston, MA: Springer US, 2001, pp. 235–250, ISBN: 978-0-306-47003-5.

[47] Weifeng Wang and Li Jiao, « Trace Abstraction Refinement for Timed Automata », *in*: *Automated Technology for Verification and Analysis*, ed. by Franck Cassez and Jean-François Raskin, Cham: Springer International Publishing, 2014, pp. 396–410, ISBN: 978-3-319-11936-6.

[48] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska, « Feature-Guided Black-Box Safety Testing of Deep Neural Networks », *in*: *CoRR* abs/1710.07859 (2017), arXiv: 1710.07859, URL: http://arxiv.org/abs/1710.07859.

[49] Longfei Wu, Haotian Chi, and Xiaojiang Du, « A Secure Proxy-based Access Control Scheme for Implantable Medical Devices », *in*: *CoRR* abs/1803.07751 (2018), arXiv: 1803.07751, URL: http://arxiv.org/abs/1803.07751.

[50] Sergio Yovine, « Kronos: A verification tool for real-time systems », *in*: *International Journal on Software Tools for Technology Transfer* 1.*1-2* (1997), pp. 123–133.

Titre : Vérification efficace de systèmes en temps réel

Mot clés : Automates temporisés, Vérification, Model-checking, Méthodes formelles, Raffinement d'abstraction

Résumé : Les automates temporisés sont souvent utilisés pour modéliser des systèmes en temps réel. Le problème d'accessibilité est notamment étudié puisqu'il permet de vérifier des propriétés de sureté mais aussi de générer des contrôleurs pour réaliser une tâche. Bien que ce problème soit déjà résolu depuis plus de 25 ans et implémenté dans plusieurs outils, nous proposons des algorithmes pour accélérer ces méthodes dans des cas particuliers. Pour le problème de sureté nous proposons des méthodes basées sur des abstractions de zones temporelles, surestimant les parties accessibles. Ces abstractions sont ensuite successivement raffinées grâce à une boucle CEGAR. Pour le problème de génération de contrôleur, nous proposons des algorithmes basées sur des exploration heuristiques et A*. Nous présentons aussi des implémentations de ces algorithmes, ainsi que des résultats sur des différents exemples.

Title: Efficient verification of real-time systems

Keywords: Timed automata, Formal verification, Model-checking, Formal methods, Abstraction refinement

Abstract: Timed automata are often used to model real-time systems. The reachability problem is of particular interest since it allows us to check for safety properties, but also build controllers for motion planning for example. Although this problem has been solved for 25 years, and implementation of these algorithms can be found in many tools, we propose some algorithms to speed up this process in particular cases. For safety problems, we propose and test abstraction based methods, that overestimate zones of a time space that could be reachable. These abstractions are then refined successively through a CEGAR loop. As for the problem of controller generation for motion planning, we propose algorithms based on heuristic search and A*. For all these algorithms we then show experimental results that we obtained with our implementation.