Formal verification at design stage of diagnosis related properties for discrete event and real-time systems

Lulu He

université
PARIS-SACLAY

Formal verification at design stage of diagnosis related properties for discrete event and real-time systems

Vérification formelle au stade de la conception de propriétés liées au diagnostic des systèmes à événements discrets et temps réel

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580,
Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique
Graduate School : Informatique et Sciences du Numérique,
Référent : Faculté des Sciences d'Orsay

Thèse préparée dans l'unité de recherche **Laboratoire Méthodes Formelles (Université Paris-Saclay, CNRS, ENS Paris-Saclay)**, sous la direction de **Philippe DAGUE**, professeur émérite, et sous le co-encadrement de **Lina YE**, maître de conférences

Thèse soutenue à Paris-Saclay, le 18 mai 2022, par

Lulu HE

Composition du jury

Sylvain CONCHON Professeur, Université Paris-Saclay, LMF	Président
Yannick PENCOLÉ Chargé de Recherche CNRS, HDR, LAAS-CNRS Toulouse	Rapporteur & Examinateur
Lakhdar SAÏS Professeur, Université d'Artois, CRIL	Rapporteur & Examinateur
Thierry JÉRON Directeur de Recherche Inria, Centre Inria Rennes – Bretagne Atlantique	Examinateur
Philippe DAGUE Professeur émérite, Université Paris-Saclay, LMF	Directeur de thèse

THESE DE DOCTORAT
NNT : 2022UPASG037

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Vérification formelle au stade de la conception de propriétés liées au diagnostic des systèmes à événements discrets et temps réel

Mots clés : vérification formelle, diagnostic de panne, diagnosticabilité, manifestabilité

Résumé : Le diagnostic de pannes est une tâche cruciale et difficile dans le contrôle automatique des systèmes complexes, dont l'efficacité dépend d'une propriété du système appelée diagnosticabilité. La diagnosticabilité décrit la propriété du système permettant de déterminer dès la phase de conception si un défaut donné se produisant en ligne sera identifiable avec certitude sur la base des observations disponibles, ce qui est une alternative aux tests qui ne peuvent que montrer la présence de défaillances sans garantir leur absence. Le problème de diagnosticabilité des systèmes à événements discrets a reçu une attention considérable dans la littérature, mais peu nombreux sont les travaux qui prennent en compte des contraintes de temps explicites lors de cette analyse. Or de telles contraintes sont naturellement présentes dans les systèmes réels et ne peuvent être négligées compte tenu de leur impact sur cette propriété. Nous avons proposé dans notre travail de master une nouvelle approche à base de SMT (Satisfiability Modulo Theories) pour vérifier la diagnosticabilité en temps borné sur les automates temporisés. L'idée est d'encoder en SMT la condition nécessaire et suffisante de diagnosticabilité. Afin d'améliorer l'efficacité de notre méthode (le problème est PSPACE-complet), nous en proposons à présent une extension incrémentale fondée sur l'utilisation de sur- et sous-approximations paramétrées dans une généralisation de la méthode CEGAR (raffinement d'abstraction guidé par un contre-exemple). Nous montrons l'amélioration apportée au travers de résultats expérimentaux.

Néanmoins, la diagnosticabilité est une propriété assez forte, qui nécessite généralement un nombre élevé de capteurs. Par conséquent, il n'est pas rare que le développement d'un système diagnosticable soit trop coûteux. Afin de garantir dès la conception un certain niveau de sûreté de fonctionnement de manière économique et efficace, nous proposons deux approches.

La première consiste à concevoir des systèmes à événements discrets diagnosticables en utilisant des blocs de retard. En effet, que se passe-t-il si un système se révèle comme non diagnosticable ? Une manière classique est d'ajouter des capteurs. Nous proposons une nouvelle manière non intrusive de rendre diagnosticable un système non diagnosticable en ajoutant simplement des blocs de retard sur certains événements observables, reportant ainsi leurs observations. Pour autant que nous le sachions, il s'agit de la première tentative de suppression de la non-diagnosticabilité avec des blocs de retard sans utiliser d'événements contrôlables ni modifier la structure des systèmes. Notre approche est codée dans une formule SMT, dont l'exactitude et l'efficacité sont démontrées par nos résultats expérimentaux.

La seconde consiste à analyser une nouvelle propriété du système appelée manifestabilité, qui est une exigence plus faible sur les observations du système pour avoir une chance d'identifier l'occurrence des défauts en ligne et peut être vérifiée au stade de la conception. Intuitivement, cette propriété garantit qu'un système défectueux ne peut pas toujours apparaître sain, c'est-à-dire qu'il a au moins un comportement futur après l'apparition d'un défaut qui se distingue par l'observation de tous les comportements normaux. Nous définissons d'abord la manifestabilité des automates à états finis pour les systèmes à événements discrets et proposons un algorithme de complexité PSPACE pour la vérifier automatiquement et prouvons que le problème de vérification de la manifestabilité lui-même est PSPACE-complet. Les résultats expérimentaux montrent la faisabilité de notre algorithme d'un point de vue pratique. Ensuite, nous définissons la manifestabilité des systèmes temps-réel modélisés par des automates temporisés en tenant compte des contraintes de temps, et étendons notre approche pour vérifier la manifestabilité de ces systèmes, prouvant qu'elle est indécidable en général mais, sous certaines conditions restreintes, devient PSPACE-complet. Enfin, nous encodons cette propriété dans une formule SMT, dont la satisfaisabilité témoigne de la manifestabilité, avant de présenter des résultats expérimentaux montrant le passage à l'échelle de notre approche.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title: Formal verification at design stage of diagnosis related properties for discrete event and real-time systems

Keywords: formal verification, fault diagnosis, diagnosability, manifestability

Abstract: Fault diagnosis is a crucial and challenging task in the automatic control of complex systems, whose efficiency depends on a system property called diagnosability. Diagnosability describes the system property allowing one to determine at design stage whether a given fault occurring online will be identifiable with certainty based on the available observations, which is an alternative to testing that can only show the presence of failures without guaranteeing their absence. The diagnosability problem of discrete event systems has received considerable attention in the literature, but little work takes into account explicit time constraints during this analysis. However such constraints are naturally present in real-life systems and cannot be neglected considering their impact on this property. We proposed in our master work a new SMT (Satisfiability Modulo Theories)-based approach to verify bounded time diagnosability on timed automata. The idea is to encode in SMT the necessary and sufficient condition for diagnosability. In order to improve the efficiency of our method (the problem is PSPACE-complete), we propose now an incremental extension of it based on the use of parameterized over- and under-approximations generalizing the CEGAR (CounterExample-Guided Abstraction Refinement) method. We show the improvement provided through experimental results.

Nevertheless, diagnosability is a quite strong property, which generally requires a high number of sensors. Consequently, it is not rare that developing a diagnosable system is too expensive. In order to guarantee from design an adequate level of safety in an economical and efficient way, we propose two approaches.

The first one consists in designing diagnosable discrete event systems by using delay blocks. Indeed, what if a system is revealed as non-diagnosable? One classical way is to add sensors. We propose a new non-intrusive way to make diagnosable a non-diagnosable system by merely adding delay blocks on some observable events, thus deferring their observations. As far as we know, this is the first attempt to remove non-diagnosability with delay blocks without using controllable events or changing the structure of systems. Our approach is encoded into an SMT formula, whose correctness and efficiency are demonstrated by our experimental results.

The second one consists in analyzing a new system property called manifestability, that is a weaker requirement on system observations for having a chance to identify online fault occurrence and can be verified at design stage. Intuitively, this property makes sure that a faulty system cannot always appear healthy, i.e., has at least one future behavior after fault occurrence observably distinguishable from all normal behaviors. We first define the manifestability of finite state automata for discrete event systems and propose an algorithm with PSPACE complexity to automatically verify it and prove that the problem of manifestability verification itself is PSPACE-complete. The experimental results show the feasibility of our algorithm from a practical point of view. Then we define the manifestability of real-time systems modeled by timed automata by taking into account time constraints, and extend our approach to verify manifestability for these systems, proving that it is undecidable in general but, under some restricted conditions, becomes PSPACE-complete. Finally we encode this property into an SMT formula, whose satisfiability witnesses manifestability, before presenting experimental results showing the scalability of our approach.

Acknowledgements

I want to acknowledge financial support of this thesis during the past three years by a scholarship from the French government, in particular from the Science and Technologies of Information and Communication doctoral school of Paris-Saclay. Besides, I also want to thank the France nation for hosting and taking care of me. Special gratitude to the Paris city: its beautiful scenery, delicious food, interesting activities and lovely citizens have left a wonderful impression on me. They helped me to have a nice leisure time and to get through the difficult moments in the working process.

I would like to gratefully acknowledge the reviewers of this thesis, Professor Yannick Pencolé and Professor Lakhdar Saïs for the valuable time they have allocated in order to evaluate the quality of this work, despite their busy schedules. And thanks to Professor Sylvain Conchon and Professor Thierry Jéron for having accepted to be members of my thesis committee.

I would like to express my special thanks of gratitude to my supervisor Professor Philippe Dague and co-supervisor Associate Professor Lina Ye who gave me the golden opportunity to focus on this wonderful project and also helped me in doing a lot of research. Moreover, they were always kind, encouraging and supportive. They played a role of advisor, boss, psychologist, teacher and good friend, more than just a supervisor.

It was extremely difficult to live far away from my family in China. Sincere thanks from my deep heart to my dear friends in France, including: Yiran Zhang, Wenbo Zhou and Feng Jin.etc. thank you all of you for the help and companionship. Wonderful moments together will last forever.

Finally, all my deep love to my parents for the endless of love and support and I sincerely appreciate to be your daughter. How lucky I am to be your little daughter. Now it is an end for my student career lasting for more than 20 years. During the past 20 years, I started from learning how to speak and write to acquiring knowledge and skills. And now I try to find a problem, analyze it critically and create a new small world. What a wonderful journey. Life is a journey not a destination. I will keep moving.

Contents

1 General Introduction **11**
- 1.1 Motivation . . . 11
- 1.2 Contributions . . . 12
- 1.3 Thesis Organization . . . 15

2 Preliminaries and State of the Art **17**
- 2.1 Introduction to Fault Diagnosis . . . 17
- 2.2 Modeling Formalism . . . 19
  - 2.2.1 Discrete Event Systems . . . 20
  - 2.2.2 Real-Time Systems . . . 22
- 2.3 Diagnosability . . . 24
  - 2.3.1 Diagnosability Checking in DESs . . . 25
  - 2.3.2 Diagnosability Checking in RTSs . . . 26
- 2.4 SAT Problem . . . 28
  - 2.4.1 SAT Algorithms and Heuristics . . . 30
  - 2.4.2 SAT-based Diagnosability Encoding for DESs . . . 32
- 2.5 SMT Problem . . . 35
  - 2.5.1 Background . . . 36
  - 2.5.2 SMT Algorithm . . . 37
  - 2.5.3 SMT Solver . . . 39
  - 2.5.4 SMT-based Diagnosability Encoding for RTSs . . . 40
  - 2.5.5 Encoding Timed Automaton . . . 41
  - 2.5.6 Encoding Bounded Diagnosability . . . 42
- 2.6 CEGAR and RECAR Algorithms . . . 45
  - 2.6.1 CEGAR Algorithm . . . 46
  - 2.6.2 RECAR Algorithm . . . 49

3 An Approximation-based Incremental SMT-based Approach to Diagnosability Analysis of Real-Time Systems **51**
- 3.1 Motivation . . . 51
- 3.2 CEGAR-over for Bounded Diagnosability Analysis of RTS . . . 52
- 3.3 CEGAR-under for Bounded Diagnosability Analysis of RTS . . . 55
- 3.4 A RECAR-like Approach for Bounded Diagnosability Analysis of RTS . . . 57
- 3.5 Encoding RECAR-like Approach . . . 59
  - 3.5.1 Pre-processing . . . 59
  - 3.5.2 Encoding Changeable Parameters . . . 60
- 3.6 Experiments . . . 61
- 3.7 Results and Discussion . . . 64

3.8 Conclusion . . . 66

4 Designing Diagnosable Discrete Event Systems by using Delay Blocks 69
4.1 Motivation/Introduction . . . 69
4.2 Designing Diagnosable Systems with Delay Blocks . . . 70
4.2.1 Automata with Delay Blocks (ADB) . . . 72
4.2.2 Unfolding FSA into Flow Network . . . 74
4.2.3 Encoding Min-cut . . . 75
4.2.4 Diagnosability Conditions . . . 79
4.3 Implementation and Validation . . . 79
4.4 Related Work . . . 80
4.5 Conclusion . . . 81

5 Manifestability Property and its Verification 83
5.1 Motivation/Introduction . . . 83
5.1.1 Motivating Example . . . 84
5.2 Manifestability Analysis for Discrete Event Systems . . . 85
5.2.1 Manifestability Property for DESs . . . 85
5.2.2 Manifestability Verification . . . 90
5.2.3 Experimental Results . . . 97
5.3 Manifestability Analysis for Real-Time Systems . . . 98
5.3.1 Motivation/Introduction . . . 98
5.3.2 Manifestability Property for RTSs . . . 99
5.3.3 Undecidability and Decidability Results . . . 100
5.3.4 Encoding Bounded Manifestability . . . 103
5.3.5 Experimental Results . . . 108
5.4 Related Work . . . 109
5.5 Comparison with Opacity . . . 110
5.6 Conclusion . . . 114

6 Conclusion 115
6.1 Thesis Overview . . . 115
6.1.1 Diagnosability . . . 115
6.1.2 Manifestability . . . 116
6.2 Future Work . . . 117

A French synthesis 119

B Publications 123

C Software 125

List of Figures

2.1 Model-based diagnosis. 18
2.2 A system example modeled by a finite automaton. 21
2.3 A system example modeled by a timed automaton. 24
2.4 A schematic overview of Z3. 40
2.5 The CEGAR framework with over-approximation. 48
2.6 The CEGAR framework with under-approximation. 48
2.7 The RECAR framework 50
2.8 The RECAR algorithm 50

3.1 The CEGAR-over framework for diagnosability checking. 53
3.2 Over-approximation refinement process. 53
3.3 The CEGAR-under framework for diagnosability checking. 55
3.4 Under-approximation refinement process. 55
3.5 The RECAR-like framework for diagnosability checking. 58

4.1 Diagnoser of the system in Figure 2.2 71
4.2 Refined diagnoser of the system in Figure 2.2 71
4.3 Twin plant of the system in Figure 2.2 71
4.4 Critical twin plant of the system in Figure 2.2 72
4.5 An example of ADB 74
4.6 Unfolding the critical twin plant of Figure 4.4 75

5.1 A simplified HVAC system. 84
5.2 A system model (top) and its diagnoser (bottom) 86
5.3 Fault diagnoser (top) and its refined version (bottom) for Example 11 91
5.4 Normal diagnoser (top) and its refined version (bottom) for Example 11 92
5.5 The pair verifier for the system in Example 11 93
5.6 A real-time system model TA. 100
5.7 The fault pair verifier V_A^F for the system whose model is depicted in Figure 4.1 (top); the fault diagnoser D_A^F (bottom). 101

List of Tables

3.1 Experimental results before optimization 60
3.2 Experimental results after optimization 60
3.3 Experimental results for CEGAR-over 62
3.4 Experimental results for CEGAR-under 63

4.1 Experimental results 80

5.1 Experimental results of manifestability checking for DESs 97
5.2 Experimental results of manifestability checking for SS-DTA 109

1 - General Introduction

1.1 . Motivation

Safety issues are essential in the development of complex systems. Computer scientists agree on the fact that it is preferable to verify safety-critical systems during the design stage before running them, which is an alternative to testing that can only show the presence of failures without guaranteeing their absence. Hence, automated formal verification has emerged as a promising useful complement. Suitable also for the classical safety properties, this thesis is about ensuring at design stage the possibility of performing later on-line diagnosis, particularly by formally performing diagnosability and manifestability (a new property we introduce) analysis in discrete event systems and real-time systems using a logic based approach, in particular using SMT solver.

Fault diagnosis [74, 91, 43, 18] is a crucial and challenging task in the automatic control of complex systems, whose efficiency depends on a system property called diagnosability. Diagnosability describes the capability of a partially observable system to determine with certainty whether a fault has effectively occurred based on a sequence of observations. The diagnosability problem of discrete event systems has received considerable attention in the literature since its introduction [82]. However, up to now little work takes into account explicit time constraints during this analysis, which are however naturally present in real-life systems and thus cannot be neglected considering their impact on this property. Therefore, in my master research, I focused on verifying diagnosability of real-time systems and proposed a new SMT-based approach to analyse it. Now, in order to improve the efficiency of our method, we propose an incremental approach based on approximations of the problem.

It is known from literature that the diagnosability verification problem of discrete event systems is in the class P for finite state automata and becomes PSPACE-complete for timed automata. Combining Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) technologies, which have proved to be a very successful practical approach to solve NP-complete problems, have achieved good results in this case. Nevertheless, scalability is not guaranteed and too large automata cannot be handled. As Recursive Explore and Check Abstraction Refinement (RECAR) [63] appears promising for dealing with beyond NP problems, in particular PSPACE-complete problems, we propose to verify diagnosability of real-time systems efficiently by taking advantage of the RECAR framework, which requires defining over- and under-approximations and their refinements.

Diagnosable systems play a key role in automatic control and complex systems, and it is important to design a diagnosable system, which will prove to be substantially convenient for subsequent system diagnosis. One aim of diagnosability

verification is to ensure some diagnosis related properties of the system at the design stage. However, it is not easy to ensure that a designed system is diagnosable. Usually, when the system is revealed as non-diagnosable, one classical way is to just redesign a new system based on results of diagnosability analysis, in particular add new sensors, but again this is in general too expensive and may require a lot of iterations. Another way relies on controllable events that may constrain the system behaviors such that the allowed behaviors are diagnosable, which is not always feasible in practice. Up to now, most work has focused on diagnosability verification of the system, while little work has considered designing a diagnosable system economically. Thus, it is interesting and meaningful to focus on how to transform non-diagnosable systems into diagnosable ones.

The study of diagnosability of discrete event systems and real-time systems respectively shows that diagnosability, characterizing whether one can distinguish with certainty faulty behaviors from normal ones based on sequences of observable events emitted from the system, is however a quite strong property that generally requires a high number of sensors. Consequently, it is not rare that developing a diagnosable system is not practically feasible. In order to achieve a trade-off between the cost, i.e., a reasonable number of sensors, and the possibility to observe a fault manifestation, we analyze a new system property called manifestability that represents the weakest requirement on observations for having a chance to identify online fault occurrences and can be verified at design stage. Intuitively, this property makes sure that a faulty system cannot always appear healthy, i.e., has at least one future behavior after fault occurrence observably distinguishable from all normal behaviors.

1.2 . Contributions

Our contributions are threefold, as follows:

- **Approximation-based diagnosability analysis of real-time systems** (chapter 3). We use the CounterExample-Guided Abstraction Refinement (CEGAR) [36] framework and its generalization RECAR to optimize diagnosability checking algorithm for timed automata. Since diagnosability verification problem is PSPACE-complete for timed automata and RECAR appears promising for dealing with beyond NP problems, in particular PSPACE-complete problems, we take the advantage of the RECAR framework and propose a RECAR-like approach for the diagnosability problem, which improves the efficiency of diagnosability verification of real-time systems. In this part our contributions are as follows:
  - We apply CEGAR-over (i.e., CEGAR using over-approximations) to time bounded diagnosability verification of real-time systems by defining three changeable parameters for the approxmations.

- We propose three methods to apply CEGAR-under (i.e., CEGAR using under-approximations) to bounded diagnosability verification of real-time systems. The first one controls two changeable parameters. The second one replaces observation equivalence checking by the weaker and easier to prove semi-equivalence checking. And the last one reduces the size of the formula to check for satisfiability by selecting only a part of it.
- We propose a RECAR-like algorithm alternating CEGAR-over and CEGAR-under defined above.
- We encode our RECAR-like algorithm in SMT formulas and show how to realize changeable parameters by selectors in SMT. Meanwhile, we propose a method for pre-processing the transitions constituting the system that can effectively reduce the search space, which is also applicable to direct bounded diagnosability verification without the use of CEGAR.
- To show the feasibility of our algorithm, we realize a number of experiments on different benchmarks and give some conclusions on the basis of the experimental results.

- **Designing diagnosable discrete event systems by using delay blocks** (chapter 4). In order to design a diagnosable discrete event system, we propose a new non-intrusive approach by merely deferring some observable events while keeping the original system structure. More precisely, we use finite automata with delay blocks (ADB) to eliminate all critical pairs in the twin plant by taking care to not create new ones and then every faulty trace can be observably distinguished from normal traces. Our contributions to the design of diagnosable DESs are as follows:
  - We redefine a simpler version of automaton with delay blocks (ADB) by using its deferral aspect which is enough to be applied to our problem.
  - In order to efficiently eliminate all pairs violating diagnosability by adding the fewest delay blocks possible, we calculate the minimum number of corresponding transitions in normal trajectories according to the max-flow min-cut theorem, encoded in SMT. This is done in a way such that every path in the normal diagnoser that constitutes a critical pair will change its observations order and then, after synchronization with the fault diagnoser, all previous critical pairs will be eliminated.
  - We analyze the scope of our approach by characterizing the systems for which it is applicable.
  - We present experimental results on benchmarks to demonstrate the efficiency and correctness of our approach.

- **Fault manifestability analysis for discrete event and timed systems** (chapter 5). On the basis of our previous research, we could analyze diagnosability of systems modeled as Timed Automata (TA) based on SMT and further design more secure real-time systems. However, it is costly in practice since it usually requires a large number of sensors. Actually, diagnosability requires that all of future behaviors of all fault occurrences should be distinguishable from all normal behaviors, which is a strong property and sensor demanding. It is why we introduced a new system property, called manifestability (resp., strong manifestability), that requires only that at least one future behavior after at least one fault occurrence (resp., after all fault occurrences) observably distinguishes from all normal behaviors. It is in fact the weakest property to require from the system to have a chance to identify the fault occurrence. With the assumption that no behavior described in the model has zero probability, the fault will then necessarily show itself if this property holds, i.e., the system cannot always appear healthy when a fault occurs in it. Obviously, one has to continue to rely on diagnosability for online safety requirements, i.e., for those faults which may have dramatic consequence if they are not surely detected when they occur, in order to trigger corrective actions. However, for all other faults that do not need to be detected at their first occurrence, manifestability checking, which is cheaper in terms of sensors needed, is enough under the probabilistic assumption above. In chapter 3, we first give the definition of the (strong) manifestability for finite automata and its characterization by a sufficient and necessary condition as language equivalence. And we prove that the manifestability problem itself is a PSPACE-complete problem. Furthermore, the correctness and efficiency of the algorithm are also revealed by our implementation with the SMT solver Z3 and experimental results. Additionally, we extend this work to real-time systems modeled by timed automata with the following contributions.

  - We redefine (strong) manifestability property for timed automata that takes into account time constraints in an explicit way and we provide a sufficient and necessary condition to check it.
  - We prove that the manifestability problem for timed automata is undecidable by reducing to it the undecidable inclusion problem of languages of timed automata. We also study a subclass of timed automata by providing corresponding conditions, under which the manifestability problem becomes decidable (PSPACE-complete).
  - For those decidable cases, we propose to encode this problem in an SMT formula, whose satisfiability witnesses manifestability.
  - We also provide some preliminary experimental results for this SMT-based algorithm to check manifestability for timed automata, which

shows feastibility and reasonable scalability.

1.3 . Thesis Organization

This thesis is organized into six chapters, including this first introduction chapter and the last conclusion and future work chapter. Chapter 2 reviews the state of the art mainly about the diagnosability checking methods for discrete event systems (modeled as automata) and real-time systems (modeled as timed automata), by using SAT and SMT, respectively, whose principles are reminded. Then we introduce the CEGAR and RECAR frameworks that we will use in Chapter 3 to optimize diagnosability verification for real-time systems. Chapters 3 to 5 present our contributions. In chapter 3, we present our new algorithm to optimize diagnosability verification problem for timed automata by taking advantage of the RECAR framework making use of over- and under-approximations of the problem. In chapter 4, we propose a new non-intrusive way to make a non-diagnosable system diagnosable by using automata with delay blocks (ADB). In chapter 5 we consider another system property called manifestability, which is weaker than diagnosability, and we provide the methods for verifying manifestability for both automata and timed automata. In Chapter 6, we recapitulate our contributions, point out the limitations of the present work and suggest new ideas for future research.

2 - Preliminaries and State of the Art

In this chapter we provide the preliminaries required for this thesis, which will be referenced in the next chapters whenever needed. It contains the formal definitions for the system model that we adapted in our study, such as discrete event systems modeled with finite state machines (FSMs) and real-time systems with timed automata. We recall the diagnosability problem definition and review how it is addressed in the literature. The basics to understand satisfiability (SAT) and satisfiability modulo theories (SMT) will follow, then we recall the diagnosability checking for discrete event systems and real-time systems and show how they have been used to encode the diagnosability problems in SAT and SMT respectively. Finally, we introduce the theory of counterexample-guided abstraction refinement (CEGAR) and of recursive explore and check abstraction refinement (RECAR), which are the important theoretical basis I used in my thesis.

2.1 . Introduction to Fault Diagnosis

Fault diagnosis is an action of identifying a malfunctioning system based on observing its behavior. In the past few decades, science and technology have developed rapidly. With the increase of the human requirements, various devices have become more complex. While enjoying the convenience of these devices, we are also faced with losses due to equipment failures. Therefore, fault diagnosis is an important domain that will be always needed to ensure the safe availability of these systems. It is an important branch of Artificial Intelligence (AI) and has received a lot of attention from scholars.

There are four traditional fault diagnosis methods: fault tree diagnosis, redundancy diagnosis, expert system diagnosis, and model-based diagnosis (MBD). The first three methods were earlier proposed, which generally have the following drawbacks: low flexibility, strong equipment dependency and limitations in practical applications. The specific principles, advantages and disadvantages of these three fault diagnosis methods can be found in the literature [39]. In order to overcome the shortcomings of these three fault diagnosis methods, Reiter first proposed the MBD algorithm in 1987 [74], which provides an inverse deduction process based on a behavioral model of the system to be diagnosed and on consistency-based reasoning. The principle of MBD is shown in Figure 2.1. It can be seen from the figure that a model of the system to be diagnosed needs to be established first, and the model is simulated to obtain the expected behavior of the system, and then the actual behavior of the system is obtained through observation. When the behavior is inconsistent with the actual behavior, it can be considered that the system has failed. At this time, the system needs to be diagnosed and reasoning to be conducted to explain the actual behavior of the system. This is achieved

by changing the by-default assumptions of correct behavior of some components into assumptions of faulty behavior until consistency between real observation and predicted observation is restored. MBD has the following characteristics: versatility, flexibility, and reliability. Because of these characteristics, MBD is widely used in various practical applications, such as: communication systems, aerospace systems, network systems, etc. The diagnosis method used in this paper is also the MBD algorithm.

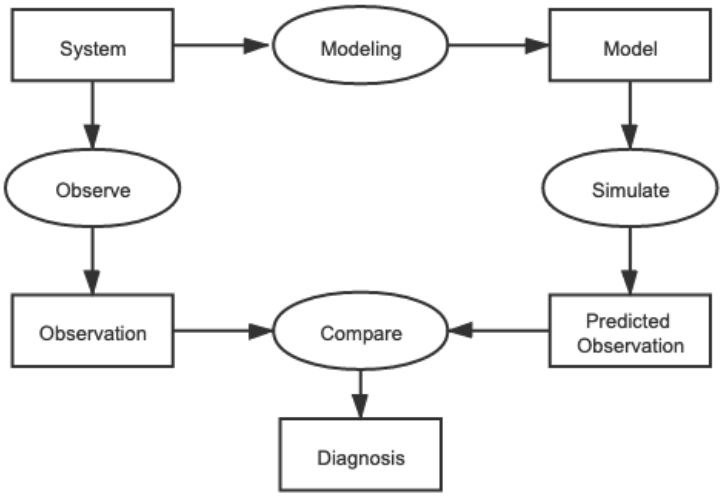

Figure 2.1: Model-based diagnosis.

In real production life, systems to be diagnosed can be divided into two main categories: static systems and dynamic systems. In static systems, the outputs are just determined by the inputs, the typical example of a static system is combinatorial circuit. With the development of MBD, the diagnosis of the static systems can no longer meet the needs of human beings. So scholars gradually began to pay attention to the diagnosis of dynamic systems, in which the outputs are determined by the inputs, state and time. Due to the uncertainty of the system, it is difficult to model dynamic systems. In order to diagnose dynamic systems, scholars have proposed as models discrete event system (DES) and real-time system (RTS), that is, time and thus the dynamics of the system are abstracted, so as to obtain a kind of system whose state evolution is driven by discrete events.

The MBD problem of DES has attracted wide attention from experts and scholars, they all try to deal with the main problem which is the compromise between the number of possible diagnoses to the faulty system and the number of observations which must be given to make the decision. DES is a modeling framework for dynamic systems whose behavior is described through discrete changes (the events), but it cannot express temporal information, which has an important impact on diagnosis. Then Alur and Dill proposed timed automaton (TA) which

is an extension of DES. In such a model, quantitative properties of delays between events can easily be expressed. The diagnosis problem always needs to cope with an explosion in the number of system model states and its complexity has actually been proved to be NP-hard.

In order to take the appropriate action in the diagnosis process, the diagnosis question must be answered precisely. However, the diagnosis decision maybe always uncertain, and thus running a diagnosis algorithm may not be accurate. For example, in the same system, the diagnostic results may be divergent because of different observations provided by different sets of sensors or at different times. This uncertainty raises the problem of diagnosability which is an essential property to be ensured at the design stage of modeling system. It simply means the ability to get the precise diagnosis. After that, the MBD will be used in applications to explain any anomaly, with a guarantee of correctness and precision, at least for each anticipated fault in the model.

2.2 . Modeling Formalism

In this section, we introduce a way to model different systems. As the name implies, MBD firstly needs to establish a model of the system to complete its diagnosis. There are three main types of modeling for DES: process algebra, Petri net and finite state automaton (FSA). Then, in order to express temporal information in the system, we also used timed automaton modeling real-time system. In fault diagnosis with process algebra, the DES is represented as two models: structural model and behavioral model. Then the system is finally diagnosed by evaluating the process algebra in terms of performance [37]. Petri net and FSA are based on a state transition structure, that is, in each state of the system, a specific event drives the system to a next state. A Petri net is composed of three elements: places, transitions, and directed edges for connections, each place can contain multiple marks. If all the fault places of the Petri net do not contain any mark, then the system is normal, otherwise the system is faulty.

FSA is the most classic DES modeling method (used in particular in the foundations of diagnosability analysis proposed by Sampath et al. in 1995 [82]), which assumes that the system behavior and fault are known in advance. This formalism abstracts away from time, retaining only the sequencing of events. In the linear time model, it is assumed that an execution can be completely modeled as a sequence of states or system events, called an execution trace (or just trace). The behavior of the system is a set of such execution traces. When the systems are finite-state, as many are, we can use finite automata, leading to effective constructions and decision procedures for automatically manipulating and analyzing system behavior.

Although the decision to abstract away from quantitative time has many advantages, often, it is necessary to consider real-time aspects: quantitative information

about time elapsing has to be handled explicitly. This can be the case when describing a particular behavior (for instance, a time-out) or stating a complex property (for example, "the alarm has to be activated *within at most 5 time units* after a problem has occurred"). In 1991, Alur and Dill have proposed *timed automata* as a model to represent the behavior of real-time systems [3]. This formalism extends classical automata with a set of real-valued variables - called clocks - that increase synchronously with time, and associates guards (specifying when, i.e. for which values of the clocks, the transition can be performed) and update operations (to be applied when the transition is performed) with every transition. Thanks to these clocks, it becomes possible to express constraints over delays between two transitions.

There are sensors with each FSA and TA to monitor events occurring in the system. Events that can be sensed by the sensor are called observable events, those that cannot be sensed are called unobservable events. Because the sensors can directly monitor observable events, it is usually assumed that the fault events which need to be diagnosed are unobservable events. When a fault event occurs, it is assumed that the system enters a fault state and cannot recover by itself. The modeling methods in our study are FSA and TA.

2.2.1 . Discrete Event Systems

Most of the systems in real life are dynamic. Therefore, the diagnosis of dynamic systems has attracted wide attention of scholars. Because it is difficult to diagnose dynamic systems, scholars abstract dynamic systems at discrete points in time as DES, and the diagnosis of the system is performed from a model of it, as a DES. This part mainly introduces the basic concepts of DES that will be used in particular for diagnosability and manifestability checking of the system.

Definition 1 (Discrete Event System) *A discrete event system is a process that is driven by (in general) asynchronous discrete events leading to the evolution of discrete states in the system.*

DES is a bridge between static and dynamic systems: there is no explicit information about real-time but discrete events are ordered as if they happened at successive unknown discrete instants. In real life, queuing systems, communication systems, manufacturing systems, and database systems can all be expressed as DES. The modeling method of DES studied in this paper is FSA.

Definition 2 (Finite State Automaton) *A finite state automaton is a four-tuple:* $G = (Q, \Sigma, \delta, q_0)$*, where:*

- Q *is a finite set of states;*
- Σ *is a finite set of events;*

- $\delta \subseteq Q \times \Sigma \times Q$ *is a finite set of transitions;*
- q_0 *is the initial state.*

The set of events Σ is divided into three disjoint parts: $\Sigma = \Sigma_o \uplus \Sigma_u \uplus \Sigma_f$, where Σ_o is the set of observable events, Σ_u the set of unobservable normal events and Σ_f the set of unobservable fault events. For representing a transition from state $q_1 \in Q$ to state $q_2 \in Q$ with event $\sigma \in \Sigma$, we will use indifferently the relational description $(q_1, \sigma, q_2) \in \delta$ or the functional description $q_2 \in trans(q_1, \sigma)$. Σ^* is the set of finite-length event sequences composed of events in Σ, and it is easy to extend inductively δ to $\delta \subseteq Q \times \Sigma^* \times Q$ as follows:

- $(q, \epsilon, q) \in \delta$, where ϵ is the null event.
- $(q, se, q1) \in \delta$ iff (if and only if) $\exists q' \in Q, (q, s, q') \in \delta$ and $(q', e, q1) \in \delta$, where $s \in \Sigma^*, e \in \Sigma$.

The second case can be simply expressed as:

$$trans(q, se) = trans(trans(q, s), e).$$

Example 1 (FSA example) *Consider the simple system model depicted in the Figure 2.2, where* $\Sigma_o = \{o1, o2\}$, $\Sigma_u = \{u\}$, $\Sigma_f = \{F\}$, *and* q_0 *is the initial state.*

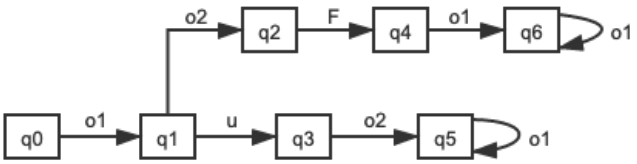

Figure 2.2: A system example modeled by a finite automaton.

Given a system G, the behavior of G is represented by the prefix closed language $L(G) \subseteq \Sigma^*$, which is the set of words produced by G:

$$L(G) = \{s \in \Sigma^* \mid \exists q \in Q, (q_0, s, q) \in \delta\}.$$

In the following, we call a word from $L(G)$ a trajectory in the system G and a sequence $q_0\sigma_0q_1\sigma_1 \ldots$ a path in G, where for all i, $(q_i, \sigma_i, q_{i+1}) \in \delta$, whose label $\sigma_0\sigma_1$... is a trajectory in G. Given $s \in L(G)$, we denote the length of s by $|s|$ and we denote the post-language of $L(G)$ after s by $L(G)/s$, formally defined as:

$$L(G)/s = \{t \in \Sigma^* \mid s.t \in L(G)\}.$$

In the definitions of diagnosability and manifestability, we will need the operation of $projection$ that consists in removing unobservable events from a trajectory.

Definition 3 (Projection) *The projection $P : \Sigma^* \rightarrow \Sigma_o^*$ on observable events is defined as $P(\epsilon) = \epsilon$, and, for any $e \in \Sigma$, $s \in \Sigma^*$, $P(se) = P(s)P(e)$, where:*

$$P(e) = \begin{cases} e & if\ e \in \Sigma_o \\ \epsilon & otherwise \end{cases} \tag{2.1}$$

The elements of $P(L(G))$ are called observed trajectories. The inverse projection for trajectories is defined by $P^{-1}(s_o) = \{s \in L(G) \mid P(s) = s_o\}$ for $s_o \in P(L(G))$. Two trajectories having same observation are called observably equivalent.

2.2.2 . Real-Time Systems

Timed automata (TA) have been proposed by R.Alur and D.Dill in the 1990s [3] as a model for real-time systems. A timed automaton is an extended classical finite automaton which can manipulate clocks, evolving continuously and synchronously with absolute time. Each transition of such an automaton is labeled by a constraint over clock values (also called guard), which indicates when the transition can be fired, and a set of clocks to be reset when the transition is fired. Each location is constrained by an invariant, which restricts the possible values of the clocks for being in the state, which can then enforce a transition to be taken. The time domain can be either discrete as $\mathbb{N}$, the set of non-negative integers, or dense as $\mathbb{Q}_+$, the set of non-negative rationals, or $\mathbb{R}_+$, the set of non-negative real numbers. In this study, we will consider dense time. TA constitute a theory for modeling and verifying real-time systems. A TA is essentially a finite automaton, thus with a finite set of states and a finite set of labeled transitions between them, extended with a finite set of real-valued variables modeling $clocks$. During a run of a TA, clock values are initialized with zero when starting in the initial state, and then are increased all with the same speed. Clock values can be compared to constants or between them. These comparisons form guards that may enable or disable instantaneous transitions and by doing so constrain the possible behaviors of the TA. Furthermore, clocks can be also reset to zero on some of the transitions.

Before introducing the formal definition of TA, we first give the set of possible clock constraints considered in this paper. The time constraints called diagonal constraints are formally described by:

$$g ::= true \mid x \bowtie c \mid x - y \bowtie c \mid g \wedge g$$

where x, y are clock variables, c is a constant and $\bowtie\ \in \{<, \leqslant, =, \geqslant, >\}$.

Note that a TA allowing such clock constraints is exponentially more concise than its classical variant with only diagonal-free constraints (where the comparison can be done only between a clock value and a constant) but both have same

expressiveness. Let X be a finite set of clock variables. A clock valuation over X is a function $v : X \rightarrow R$, where R denotes the set $\mathbb{R}_+$ of non-negative real numbers (actually, for implementation, the set $\mathbb{Q}_+$ of non-negative rational numbers is used to have an exact computer representation). Then the set of all clock valuations over X is denoted by R^X and the set of time constraints over X by $\mathbb{C}(X)$, where such a constraint is given by a collection of clock constraints. If a clock valuation v satisfies the time constraint g, then it is denoted by $v \models g$. In the following, we denote $[\![g]\!]$ the set of clock valuations that satisfy g, i.e., $[\![g]\!] = \{v \in R^X \mid v \models g\}$.

Definition 4 (Timed Automaton) *A $timed\ automaton$ (TA) is a tuple $A = (Q, \Sigma, X, \delta^X, q_0, I)$, where:*

- *Q is a finite set of states;*
- *Σ is a finite set of events;*
- *X is a finite set of clock variables;*
- *$\delta^X \subseteq Q \times \mathbb{C}(X) \times \Sigma \times 2^X \times Q$ is a finite set of transitions;*
- *$q_0 \in Q$ is the initial state;*
- *$I : Q \rightarrow \mathbb{C}(X)$ is a function that assigns invariants to states.*

Example 2 (TA example) *The figure 2.3 represents a TA obtained by adding some time constraints to the system model of figure 2.2. In this example, x is a clock variable that is used to impose certain period between events or to restrain the possible time during which one is allowed to stay in some states. For example, $(q_0, x > 1, o1, \{x\}, q_1) \in \delta^X$ means that only when the guard $x > 1$ is satisfied, i.e., the clock value is greater than 1, the event $o1$ can occur, inducing an instantaneous state change from q_0 to q_1 and simultaneously the reset to 0 of the clock x. We denote this transition also as $q_0 \xrightarrow{x>1;\ o1;\ x:=0} q_1$. Furthermore, $I(q_1) = x < 6$ imposes that we can stay in the state q_1 only when the clock value is smaller than 6. In other words, once the invariant ceases to be satisfied, one is obliged to leave the corresponding state (for readability reasons, we did not represent invariants that define also sojourn time upper bounds in states q_2, q_3, q_4).*

We call a state with a clock valuation an extension state, i.e., (q, v) with $q \in Q$ and $v \in R^X$. Let $t \in R$, the valuation $v + t$ is defined by $(v + t)(x) = v(x) + t, \forall x \in X$. Suppose $X' \subseteq X$, we denote by $v[X' \leftarrow 0]$ the valuation such that $\forall x \in X'$, $v[X' \leftarrow 0](x) = 0$ and $\forall x \in X \backslash X'$, $v[X' \leftarrow 0](x) = v(x)$. A TA gives rise to an infinite transition system with two types of transitions between extension states. One is a $time$ transition representing time passage in the same state q, during which the invariant $inv = I(q)$ for q should be always satisfied. The other one is a $discrete$ transition issued from a labeled transition $q \xrightarrow{g;\ \sigma;\ r} q'$

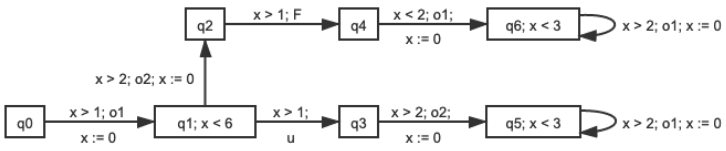

Figure 2.3: A system example modeled by a timed automaton.

for TA, associated with an event σ, which is fired (a necessary condition being that the guard g is satisfied) and should be executed instantaneously, i.e.,the clock valuation cannot be modified by the transition itself but only by the reset to 0 of those clock variables belonging to r, if any. In the following, both are denoted by $(q, v) \xrightarrow{\nu} (q', v')$, where $\nu \in \Sigma \cup R$. Thus, if $\nu \in \Sigma$, then v should satisfy the guard g in the corresponding TA transition and $v' = v[r \leftarrow 0]$ for r the clock variables reset to 0 in this transition, if any. Otherwise, if $\nu \in R$, then $q' = q$ and $v' = v + \nu$, where all of $v + t$, for $0 \leqslant t \leqslant \nu$, should satisfy the invariant inv associated to the state q.

Given A a TA, a sequence of such transitions $(q_0, v_0 = 0) \xrightarrow{\nu_1} (q_1, v_1) \ldots \xrightarrow{\nu_n} (q_n, v_n)$ is a feasible execution in A if $\forall i \in \{0, ..., n-1\}, (q_i, v_i) \xrightarrow{\nu_{i+1}} (q_{i+1}, v_{i+1})$ is either a time or a discrete transition in it. Then the word $\nu_1 ... \nu_n \in (\Sigma \cup R)^\star$ is called a *timed trajectory* or a *run*. This extends to infinite sequences and trajectories. The set of timed trajectories for A is denoted by $L(A)$. By summing up successive time periods, we can always assume that between any two successive events there is exactly one time period, i.e., periods and events alternate in a timed trajectory. For ρ a timed trajectory, we denote by $time(\rho) \in R \cup \{+\infty\}$ the total time duration for ρ, i.e., $time(\rho) = \sum_{\nu_i \in R \wedge \nu_i \in \rho} \nu_i$.

We redefine a projection operator P for TA as follows. Given a timed trajectory ρ and a set of events $\Sigma' \subseteq \Sigma, P(\rho, \Sigma')$ is the timed trajectory obtained by erasing from ρ all events not in Σ' and summing the periods between successive events in the resulting sequence. For example, if $\rho = 2\ o1\ 3\ u\ 2\ o2\ 3\ o1$, then $P(\rho, \{o1, o2\}) = 2\ o1\ 5\ o2\ 3\ o1$. In the following, Σ is divided into three disjoint parts as above and we simply denote $P(\rho)$ the projection of the timed trajectory ρ to observable events, i.e., $P(\rho) = P(\rho, \Sigma_o)$.

2.3 . Diagnosability

Diagnosability, resp. bounded diagnosability, of the considered systems is a property defined to verify the possibility to distinguish any possible faulty behavior in the system from any other behavior without this fault (i.e., correct or with a different fault), resp. within a finite number of transitions (for a FSA) or a finite time (for a TA) after the occurrence of the fault. A fault is diagnosable, resp. with

bound, if it can be surely identified from the partial observation available, resp. within a given number of transitions or a given delay after its occurrence. The first introduction to the notion of diagnosability was by Sampath et al. in 1995 [82].

2.3.1 . Diagnosability Checking in DESs

Following other studies about diagnosability in DES, we adopt the following set of assumptions to hold in all this thesis.

Assumption 1 (Living system) *The DES G is $live$, i.e., from any state, there is at least one transition issued from this state.*

This assumption of liveliness of the language $L(G)$ ensures that the post-language of $L(G)$ after any finite trajectory is never empty, so contains arbitrarily long sequences.

Assumption 2 (Observably living system) *The DES G is $observably\ live$, i.e., any infinite trajectory has infinitely many occurrences of observable events.*

This means that there is no cycle made up only of unobservable events. As studying diagnosability relies on the observations, so accepting infinite behaviors of the system without getting any observation would make the study meaningless.

Intuitively, a predefined fault is considered as diagnosable if one can be sure about its occurrence after sufficient observations, which can be formally defined as follows [82], where s^F denotes a trajectory s that ends with the fault F.

Definition 5 (Diagnosability of FSA) *Given a FSA G and a fault $F \in \Sigma_f$:*

1. *given $k \in \mathbb{N}$, F is k-$diagnosable$ in G iff*

$$\forall s^F \in L(G), \forall t \in L(G)/s^F, (|t| \geqslant k \Rightarrow \forall p \in L(G), (P(p) = P(s^F.t) \Rightarrow F \in p)).$$

2. *F is $diagnosable$ in G iff*

$$\exists k \in N \text{ such that } F \text{ is } k\text{-}diagnosable \text{ in } G.$$

We denote the $length\ from$ (the first occurrence of) $fault\ F\ in\ p$ by $length(p, F) = |t|$, where $p = sFt$ with $F \notin s$. The above definition states that for each trajectory s^F in G, for each t that is an extension of s^F with at least k events (for k-diagnosability) or sufficient events (for diagnosability), every trajectory p in G that is observationally equivalent to $s^F.t$ should contain in it F. In other words, the k-diagnosability (resp., diagnosability) checking consists in verifying the non-existence of a pair of trajectories p and p' constituting what is called a k-$critical\ pair$ (resp., a $critical\ pair$) according to the following definition [72].

Definition 6 *(Critical pair) Given a FSA G, the considered fault F and $k \in \mathbb{N}$, two trajectories p, $p' \in L(G)$ are called a $k\text{-}critical\ pair$, denoted by $p \not\sim_k p'$, if the following conditions are satisfied: 1) p contains F and p' does not; 2) $length(p, F) \geqslant k$; 3) $P(p) = P(p')$. They are called a $critical\ pair$, denoted by $p \not\sim p'$, if the following conditions are satisfied: 1) p contains F and p' does not; 2) p and p' are infinite; 3) $P(p) = P(p')$.*

A $k\text{-}critical\ pair$ (resp., a $critical\ pair$) has been proven to violate Definition 5 and thus witnesses non-k-diagnosability (resp., non-diagnosability). Consider the example in Figure 2.2, where the pair of trajectories $o1.o2.F.o1^{\omega}$ and $o1.u.o2.o1^{\omega}$ is a critical pair since it satisfies the above three conditions. In other words, once we observe the sequence of events $o1.o2.o1^{\omega}$, we can never be sure about the occurrence of the fault since in this system, there does exist one trajectory containing the fault and the other without the fault, while both have exactly this same sequence of observations.

Theorem 1 *Given G a FSA, a fault F is k-diagnosable (resp., diagnosable) in G iff there is no k-critical pair (resp., critical pair) w.r.t. F in G.*

For the sake of simplicity, we conventionally assume, in this thesis, that there is only one fault F (with multiple occurrences), i.e. $\Sigma_f = \{F\}$, which can be directly extended to the case of a set of faults by applying the approach as many times as the number of faults.

The diagnosability analysis for DES has actually been proved to be polynomial in the number of states [59].

2.3.2 . Diagnosability Checking in RTSs

We make for TA the analog assumptions (the role of the length being played by the time) made for DES about the liveness of the system and the necessity to observe any infinite behavior.

Assumption 3 (Timed living system) *The TA A is $timed\ living$ (or $well\text{-}timed$), i.e., for any reachable state, there is an infinite time timed trajectory ρ ($time(\rho) = +\infty$) starting at this state.*

Assumption 4 (Timed observably living system) *The TA A is $timed\ observably\ living$, i.e., it has no time infinite execution from a reachable state without any observable event, and thus any time infinite timed trajectory has infinitely many observable event occurrences.*

This implies in particular that the system cannot stay an infinitely long time in a same state. We rephrase a useful notion, originally introduced by [94].

Definition 7 (Δ-faulty runs) *Given A a TA, let $\rho = \nu_1\nu_2\ldots$ be a faulty timed trajectory, i.e., for some $i \in \{1, ...\}$, $\nu_i = F$. Let then j be the smallest i such that $\nu_i = F$ and let $\rho' = \nu_{j+1}\ldots$. We denote the $period\ from$ (the first occurrence of) $fault\ F\ in\ \rho$ by $time(\rho, F) = time(\rho')$. If $time(\rho, F) > \Delta$, where $\Delta \in R$, then we say that more than Δ time units pass after the first occurrence of F in ρ, or, in short, that ρ is Δ-$faulty$.*

Notice that we chose in the definition greater than Δ instead of greater or equal because some technical aspects in the encoding become easier, but there is no difference in substance, as a Δ-faulty run (for $>$) is Δ-faulty (for $\geqslant$) and a Δ-faulty run (for $\geqslant$) is Δ'-faulty (for $>$) for any $\Delta' < \Delta$. Now we adapt Definition 5 to define diagnosability of TA.

Definition 8 (Diagnosability of TA) *Given a TA A and a fault F:*

1. *given $\Delta \in R$, F is Δ-$diagnosable$ in A iff*

$$\forall \rho \in L(A)\ (\rho\ \Delta\text{-}faulty \Rightarrow \forall \rho' \in L(A)\ (P(\rho) = P(\rho') \Rightarrow F \in \rho')).$$

2. *F is $diagnosable$ in A iff*

$$\exists \Delta \in R \text{ such that } F \text{ is } \Delta\text{-}diagnosable \text{ in } A.$$

Now we define the analog of a a k-critical pair (resp., critical pair) in the timed framework, called ($timed$) Δ-$critical\ pair$ (resp., $timed\ critical\ pair$):

Definition 9 *(Timed critical pair) Given a TA A, the considered fault F and $\Delta \in R$, two timed trajectories ρ, $\rho' \in L(A)$ are called a (timed) Δ-$critical\ pair$ (Δ-CP), denoted by $\rho \not\approx_\Delta \rho'$, if the following conditions are satisfied: 1) ρ' does not contain F; 2) ρ is Δ-faulty; 3) $P(\rho) = P(\rho')$. They are called a $critical\ pair$ (CP), denoted by $\rho \not\approx \rho'$, if the following conditions are satisfied: 1) ρ contains F and ρ' does not; 2) $time(\rho) = time(\rho') = +\infty$; 3) $P(\rho) = P(\rho')$.*

Note that, in a timed critical pair, ρ and ρ' are necessarily infinite. It is obvious to prove that the existence of such a pair violates the diagnosability for TA in Definition 8 (2), and the converse has been proved too by [94].

Theorem 2 *Given A a TA, F is Δ-diagnosable (resp., diagnosable) in A iff there is no timed Δ-critical pair (resp., timed critical pair) w.r.t. F in A.*

Thus, in a way similar to FSA, the diagnosability verification for TA consists in checking the non-existence of timed critical pairs. The complexity for diagnosability analysis of RTS has been proved to be P-SPACE [94]. For example, consider the system modeled by the TA of Figure 2.3. The system has both faulty behaviors (where F occurs) and normal ones. Indeed, in all behaviors, the observable events

occur in the same order, i.e., $o1.o2.o1^*$, as in the FSA without time constraints (Figure 2.2). However, in every faulty behavior, the time duration between the successive observable events $o2$ and $o1$ is smaller than 2 time units. While in every normal behavior, this duration is greater than 2. Thus, observing $o2$ and then $o1$ and measuring the duration between them, one can tell with certainty whether a fault has occurred or not. One can clearly see that adding time constraints sometimes makes a non-diagnosable system diagnosable by distinguishing temporally the two trajectories that are considered as a critical pair in the untimed setting, e.g., with different durations between two successive observable events.

2.4 . SAT Problem

This section of the state-of-the-art follows part of [57]. In logic and computer science, the Boolean satisfiability problem, also called propositional satisfiability problem or SAT problem, is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. The given Boolean formula constituted by Boolean variables and conjunction of a set of clauses from these variables, its called Conjuctive Normal Form, denoted by CNF. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is unsatisfiable. Let us define some terms below. Boolean variables, also known as propositional variables, are symbols for 0-ary predicates that take their values in $\{True, False\}$,or $\{1, 0\}$ of logical truth values. Assignments (resp. partial assignments) are evaluations of variables (resp. part of them). An assignment that satisfies a given formula is called a model of that formula. A formula that has a model is said to be satisfiable. A formula is constructed over a set of propositional variables by using the following five logical operators or connectives:

- the binary conjunction operator AND, denoted by $\wedge$,
- the binary disjunction operator OR, denoted by $\vee$,
- the unary negation operator, denoted by $\neg$,
- the binary implication operator, denoted by $\rightarrow$,
- the binary equivalence operator, denoted by $\leftrightarrow$.

Note that the operators above are not independent and that these operators can be transformed into each other, for example, if we restrict the logical formulae to only the conjunction and negation operators, the other connectives can be easily expressed using these two operators. Let Φ and Ψ be two logical formulas, and these equivalences (denoted by $\equiv$) hold as follows:

- $\Phi \wedge \Psi \equiv \neg(\neg\Phi \vee \neg\Psi)$
- $\Phi \rightarrow \Psi \equiv \neg\Phi \vee \Psi$
- $\Phi \leftrightarrow \Psi \equiv (\Phi \rightarrow \Psi) \wedge (\Psi \rightarrow \Phi)$

In satisfiability studies, in order to push negation operators to propositional variables, we keep negation, disjunction and conjunction operators. A propositional variable or its negation is called a literal, a clause is a disjunction of literals, so if there exists an assignment that sets at least one of its literal to True, this clause is satisfied by this assignment. A CNF formula, is a conjunction of clauses, so to satisfy a CNF formula we must satisfy all its clauses, in other words, one unsatisfiable clause is sufficient to make the whole CNF formula unsatisfiable. It has been shown that any logical formula can be polynomially transformed into a CNF formula while keeping the possibility of satisfaction unchanged [95].

A simple illustrative example of a SAT problem is given below.

Example 3 (SAT Instance) *Let x,y,z be propositional variables and $\Phi = (\neg x \vee y) \wedge (\neg y \vee z)$ a CNF formula. Then, verifying the possibility to satisfy Φ, denoted by $Sat(\Phi)?$, is the SAT problem for Φ.*

For the above problem, if we use any of the sat solvers, then the solver will return SAT and a set of assignments, for example, the assignment $s = 0, y = 1, z = 1$ is a possible solution, this assignment make the formula ϕ True, so we say the formula Φ is satisfiable or Φ is SAT, and conversely, if no assignment satisfies Φ, the solver returns UNSAT, in this case, we say that the formula Φ is UNSAT, or Φ is unsatisfiable.

Nowadays, SAT is widely used in a number of areas for a number of reasons. Firstly, SAT is the prototype of NP-complete problems. From an application point of view, it is sufficient to know that an NP-complete problem is hard, that there is no way to find a solution efficiently (but it may be possible to find an approximate best solution efficiently), and that NP-complete problems are interchangeable in polynomial time (i.e., you can take one NP-complete problem and solve another obtained from the first by polynomial reduction). In the field of research, many mathematical, computer, and even industrial problems can be reduced to SAT problems, which can be naturally abstracted as constraint-based problems that consist in satisfying a set of requirements, where each requirement can be satisfied in multiple ways, and where the ways in which the different requirements are satisfied may contradict each other, thus they can be easily mapped to CNF formulations. Secondly, the solving method of SAT is well developed. SAT is the oldest and best-known NP-complete problem, and many algorithms and techniques have been developed to solve it. In particular, the efficiency of the SAT solvers can be greatly increased [67] when the Conflict Driven Clause Learning (CDCL) component is available [88], and regular SAT solver competitions like [90, 7] inspired developers to provide more efficient SAT solvers. Nowadays, CDCL solvers

can handle problems with millions of clauses and hundreds of thousands of propositional variables. This is why we now find SAT applications in many areas, such as:

- Formal methods, such as Bounded Model Checking [21], Test generation, etc.
- AI, as in Planning [60], Knowledge Representation, Games.
- Design Automation, as in fault diagnosis [52, 24, 96].
- Other applications in security, bio-informatics, mathematical problems and in the core of other constraint solvers: MAXSAT, #SAT, etc.

In addition, the successful application and rapid development of SAT techniques are also due to a community of developers who have assembled a number of well-designed algorithms and heuristics that have been designed together to answer SAT questions very effectively. In our research, we have used SAT techniques to deal with the studied problem, which may be seen as an additional application to this successful technology, and have not contributed to the development of SAT techniques, so we will recall their underlying principles here and only what we think will be useful for the reader to understand our contribution.

2.4.1 . SAT Algorithms and Heuristics

The principal rule of logical reasoning also used in the SAT algorithms is the $resolution\ rule$. $Resolution$ is a process to generate a clause from two clauses, for example, given two clauses $(\neg x \vee y)$ and $(\neg y \vee z)$, the $resolution$ of these two clauses is $(\neg x \vee z)$, because $(\neg x \vee y) \wedge (\neg y \vee z)$ is satisfiable iff $(\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg x \vee z)$ is satisfiable. This is noted as: $\frac{\neg x \vee y \quad \neg y \vee z}{\neg x \vee z}$. Using the relations cited above between the logical connectives, we can read this example as: if x implies y and y implies z, then x implies z.

Then we have the following basic result: a set (conjunction) of clauses is unsatisfiable iff it is possible to produce an empty clause from it by repeatedly applying a finite number of times the resolution rule.

Unit Propagation

Unit propagation (UP) is a procedure of automated theorem proving that can simplify a set of (usually propositional) clauses, which is a special case of application of the resolution rule. The procedure is based on unit clauses, i.e. clauses that are composed of a single literal l. Because each clause needs to be satisfied, we know that this literal must be true. If a set of clauses contains a unit clause, the other clauses are simplified by the application of the two following rules:

- Every clause (other than the unit clause itself) containing l is removed (the clause is satisfied if l is);
- In every clause that contains $\neg l$ this literal is deleted ($\neg l$ cannot contribute to the satisfiability).

The application of these two rules leads to a new set of clauses that is equivalent to the old one.

Davis and Putnam algorithm (1960) and DPLL (1962)

In 1960, Davis and Putnam proposed the first algorithm for solving the SAT problem, denoted by DP60, [41]. This algorithm used simple iteration of the resolution rule, i.e., in solving the SAT problem, the resolution rule is used iteratively until either formula becomes empty, in this case, the formula is satisfiable or, if one clause becomes empty, the formula is unsatisfiable. Although this algorithm can solve the SAT problem, it requires a huge amount of memory even after optimization like deleting pure literals and shrinking clauses with unit propagation, and because of this obvious drawback, DP60 was not used for large instances.

To optimize DP60, the DPLL (Davis-Putnam-Logemann-Loveland) algorithm was introduced two years later [42], which is a complete, backtracking-based algorithm for solving the satisfiability problem for propositional logic in CNF form. DPLL was the basis for future efficient SAT and SMT solvers and many automatic theorem proving methods in first order logic. The idea of DPLL is as follows:

- First apply unit propagation as long as possible.
- If we cannot proceed by unit propagation or trivial observations then choose a variable p, introduce the cases p and $\neg p$, and for both cases go on recursively.

This recursive process must terminate since every recursive call decreases the number of variables. If "satisfiable" is returned then all involved unit clauses yield a satisfying assignment. Otherwise, it is a big case analysis yielding $\bot$ for all cases, so unsat. DPLL efficiency strongly depends on the choice of the variable. Current SAT solvers follow this scheme, combined with good heuristics for variable choice, CDCL component, restarts and other optimizations.

Incremental SAT

Incremental SAT is the effective solution of a sequence of related SAT problems. It uses information already learned to avoid repeating redundant work. A typical stand-alone SAT-solver accepts a problem as input, and outputs a model or UNSAT as result. This can be inadequate if we wish to solve many similar SAT-instances. The most obvious overhead is re-parsing the same clause set and carrying out

the same inferences over and over again. So, equipping the SAT-solver with an interface that allows the next SAT-instance to be specified incrementally from the current (solved) instance will certainly remove the parsing problem. And actually it may reduce the number of inferences too. In our study, we focus on the type of solver based on $conflict\ analysis$ and $clause\ recording$, such solver implements a DPLL-style backtracking search procedure. The idea of incremental SAT is to spend some effort on finding a "reason" for every conflict detected during the search, that can be encoded as a clause and added to the clause set. The recorded clauses will serve as a cache for the same type of conflicts in later part of the search-space of this SAT-instance, it may be useful also in later similar SAT-instances. In this respect, we add a set of hypotheses to each clause of the original formula, then when the solver gets a new input instance to test, it will first apply the assumptions in the first level of the decision to filter the learned clauses and keep only those which are consistent with the current assumptions, then it will proceed to classical test (the coding of assumptions in SMT Z3 through selectors will be used in chapter 3 and presented in subsection 3.5.2). In this way, we can ensure the soundness of the test and improve efficiency by avoiding re-parsing the same clauses.

2.4.2 . SAT-based Diagnosability Encoding for DESs

With respect to the properties we studied, a straightforward rephrasing of the definition 5 shows that in a system G with fault F, if it exists a pair of trajectories corresponding to cycles (and hence to infinite paths), which includes a faulty one and a normal one, sharing the same observable events, shows that F is non-diagnosable. This corresponds to the existence of an ambiguous (i.e., made up of pairs of states reachable respectively by a faulty path and a correct path) cycle in the product of G by itself, synchronized over observable events, which is the origin of the so-called twin plant structure introduced in [59].

In [52, 76], the authors formulated non-diagnosability test as a satisfiability problem in propositional logic for a Succinct Transition System (SLTS) which the state is defined as a valuation of a set of Boolean state variables. To do this, the authors introduced two copies of path (faulty and normal) to distinguish between an occurrence of an event in one or the other, this idea is the same as [59]. The difference lies in the fact that this approach constructs a logical formula and the task of the solver is to find a model (if any). In other words, provided with diagnosability property encoding, the (progressive and incomplete) construction of the twin plant is due to the solver and only the search space is given to it. Thus, for each possible step in the system, it may contain observable events which belong to both the faulty sequence and the normal sequence, but which must occur simultaneously. Afterwards, the authors consider n steps by making the conjunction of their local formulas to constitute a unique logical formula that represents the sequence of events occurrences through the two sequences (normal and faulty).

Finally, they provide the resulting formula to the SAT solver. The satisfiability of this formula corresponds to finding a critical pair of length at most n, i.e. the non-diagnosability of the fault when limited to a range of n steps (bounded model checking).

We recall below this encoding with the variables and the formulas used, where superscripts t refer to steps and (e_o^t) and $\left(\hat{e}_o^t\right)$ refer respectively to the faulty and correct sequences of event occurrences (corresponding states being described by valuations of $\left(a^t\right)$ and $\left(\hat{a}^t\right)$) of a pair of trajectories witnessing non-diagnosability (so sharing the same observable events represented by (e^t) and forming a cycle). The increasing of the step corresponds to the triggering of at least one transition and the extension by an event of at least one of the two trajectories. $T = \langle A, \Sigma_o, \Sigma_u, \Sigma_f, \delta, s_0\rangle$ being an SLTS, the propositional variables are thus:

- a^t and $\hat{a}^t$ for all $a \in A$ and $t \in \{0, \ldots, n\}$,
- e_o^t for all $e \in \Sigma_o \cup \Sigma_u \cup \Sigma_f, o \in \delta(e)$ and $t \in \{0, \ldots, n-1\}$,
- $\hat{e}_o^t$ for all $e \in \Sigma_o \cup \Sigma_u, o \in \delta(e)$ and $t \in \{0, \ldots, n-1\}$,
- e^t for all $e \in \Sigma_o$ and $t \in \{0, \ldots, n-1\}$.

The following formulas express the constraints that must be applied at each step t or between t and $t+1$.

- 1. The event occurrence e_o^t must be possible in the current state:

  $$e_o^t \rightarrow \phi^t \quad \text{for } o = \langle \phi, c\rangle \in \delta(e)$$

  and its effects must hold at the next step:

  $$e_o^t \rightarrow \bigwedge_{l \in c} l^{t+1} \quad \text{for } o = \langle \phi, c\rangle \in \delta(e)$$

  Where $\delta(e)$ assigns to each event a set of pairs $\langle \phi, c\rangle$, each pair represents an occurrence of the event such that precondition ϕ is given by a formula, the propositional language built on A, that has to be satisfied by the source state of the transition and effects c are given by a set of elements in L, the literals built from A, expressing the positive and negative changes of the valuation of the destination state of the transition w.r.t. the valuation of its source state. We have the same formulas with $\hat{e}_o^t$.

- 2. The present value (True or False) of a state variable changes to a new value (False or True, respectively) only if there is a reason for this change, i.e., because of an event that has the new value in its effects (so, change without reason is prohibited). Here is the change from True to False (the change from False to True is defined similarly by interchanging a and $\neg a$):

  $$\left(a^t \wedge \neg a^{t+1}\right) \rightarrow \left(e_{i_1 o_{j_1}}^t \vee \cdots \vee e_{i_k o_{j_k}}^t\right)$$

where the $o_{j_l} = \langle \phi_{j_l}, c_{j_l} \rangle \in \delta\left(e_{i_l}\right)$ are all the occurrences of events e_{i_l} with $\neg a \in c_{j_l}$. We have the same formulas with $\hat{a}^t$ and $\hat{e}^t_{i_{l o_{j_l}}}$.

- 3. At most one occurrence of a given event can occur at a given step and the occurrences of two different events cannot be simultaneous if they interfere (i.e., if they have two contradicting effects or if the precondition of one contradicts the effect of the other):

$$\neg\left(e^t_o \wedge e^t_{o'}\right) \quad \forall e \in \Sigma, \forall \left\{o, o'\right\} \subseteq \delta(e), o \neq o'$$
$$\neg\left(e^t_o \wedge e'^t_{o'}\right) \quad \forall \left\{e, e'\right\} \subseteq \Sigma, e \neq e', \forall o \in \delta(e), \forall o' \in \delta\left(e'\right), interfere(o, o')$$

  We have the same formulas with $\hat{e}^t_o$.

- 4. The formulas that connect the two event sequences require that observable events take place in both sequences whenever they take place (use of e^t for synchronization):

$$\bigvee_{o \in \delta(e)} e^t_o \leftrightarrow e^t \text{ and } \bigvee_{o \in \delta(e)} \hat{e}^t_o \leftrightarrow e^t \quad \forall e \in \Sigma_o$$

- 5. To avoid trivial cycles (silent loops with no state change) we require that at every step at least one event takes place:

$$\bigvee_{e \in \Sigma_o} e^t \vee \bigvee_{e \in \Sigma_u \cup \Sigma_f, o \in \delta(e)} e^t_o \vee \bigvee_{e \in \Sigma_u, o \in \delta(e)} \hat{e}^t_o$$

  The conjunction of all the above formulas for a given t is denoted by $\mathcal{T}(t, t+1)$. A formula for the initial state s_0 is:

$$\mathcal{I}_0 = \bigwedge_{a \in A, s_0(a)=1} \left(a^0 \wedge \hat{a}^0\right) \wedge \bigwedge_{a \in A, s_0(a)=0} \left(\neg a^0 \wedge \neg \hat{a}^0\right)$$

At last, the following formula can be defined to encode the fact that a pair of trajectories is found with the same observable events and no fault in one trajectory, but the fault F (recall that $\Sigma_f = \{F\}$) in the other, which are infinite, witnessing non-diagnosability:

$$\Phi^n = \mathcal{I}_0 \wedge \mathcal{T}(0,1) \wedge \cdots \wedge \mathcal{T}(n-1, n) \quad \wedge$$
$$\bigvee_{t=0}^{n-1} \bigvee_{o \in \delta(F)} F^t_o \quad \wedge$$
$$\bigvee_{m=0}^{n-1} \left(\bigwedge_{a \in A} \left((a^n \leftrightarrow a^m) \wedge (\hat{a}^n \leftrightarrow \hat{a}^m) \right) \right)$$

From this encoding in propositional logic, follows the result that an SLTS T is not diagnosable if and only if $\exists n \geqslant 1, \Phi^n$ is satisfiable. It is also equivalent to

$\Phi^{2^{2|A|}}$ being satisfiable, as the twin plant state number (square of the system state number) is an obvious upper bound for n.

Note that we have presented here the SAT encoding of diagnosability checking. The encoding of k-diagnosability checking is similar. Actually, we have just to replace the last conjunct in the formula Φ^n, that checks the existence of a cycle (condition for trajectories to be infinite), by a formula that encodes $length(p, F) = k$, where p is the faulty path, which is easily done by adding variables v^{Ft} counting the number of events in p after the first occurrence of F, providing thus a formula Φ^n_k.

2.5 . SMT Problem

The diagnosability problem of discrete event systems has received considerable attention in the literature from [82]. However, up to now, few work takes into account explicit time constraints during this analysis, which are however naturally present in real-life systems and thus cannot be neglected considering their impact on this property. Therefore, my research focused on verifying diagnosability of real-time systems. As linear real arithmetic is required to deal with time constraints, we used SMT to express timed automata and then analyse their diagnosability.

The state-of-the art part of this section partly follows [12], to which we refer for more details. Applications in artificial intelligence and formal methods for hardware and software development have greatly benefited from the recent advances in SAT. Often, however, applications in these fields require determining the satisfiability of formulas in more expressive logics such as first-order logic. Despite the great progress made in the last twenty years, general-purpose first-order theorem provers (such as provers based on the resolution calculus) are typically not able to solve such formulas directly. The main reason for this is that many applications do not require general first-order satisfiability, but rather satisfiability with respect to some background theory, which fixes the interpretations of certain predicate and function symbols. For instance, applications using integer arithmetic are not interested in whether there exists a nonstandard interpretation of the symbols $<$, $+$, and 0 that makes the formula $x < y \wedge \neg(x < y + 0)$ satisfiable. Instead, they are interested in whether the formula is satisfiable in an interpretation in which $<$ is the usual ordering over the integers, $+$ is the integer addition function, and 0 is the additive neutral element. General-purpose reasoning methods can be forced to consider only interpretations consistent with a background theory T, but only by explicitly incorporating the axioms for T into their input formulas. Even when this is possible, the performance of such provers is often unacceptable. For some background theories, a more viable alternative is to use reasoning methods tailored to the theory in question. This is particularly the case for quantifier-free formulas, first-order formulas with no quantifiers but possibly with variables, such as the formula above.

For many theories, specialized methods actually yield decision procedures for the satisfiability of quantifier-free formulas or some subclass thereof. This is the case, because of classical results in mathematics, for the theory of real numbers and the theory of integer arithmetic (without multiplication). In the last two decades, however, specialized decision procedures have also been discovered for a long and still growing list of other theories with practical applications. These include certain theories of arrays and of strings, several variants of the theory of finite sets or multisets, the theories of several classes of lattices, the theories of finite, regular and infinite trees, of lists, tuples, records, queues, hash tables, and bit-vectors of a fixed or arbitrary finite size.

The research field concerned with the satisfiability of formulas with respect to some background theory is called Satisfiability Modulo Theories (SMT), which generalizes SAT by adding equality reasoning, arithmetic, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of formulas in these theories. SMT solvers enable application of bounded model checking to infinite systems. They have numerous applications in theorem proving and other domains such as real-time scheduling, temporal or metric planning, and test-case generation. In analogy with SAT, SMT procedures (whether they are decision procedures or not) are usually referred to as SMT solvers. The roots of SMT can be traced back to early work in the late 1970s and early 1980s on using decision procedures in formal methods by such pioneers as Nelson and Oppen, Shostak, and Boyer and Moore [70, 69]. Modern SMT research started in the late 1990s with various independent attempts [6, 51] to build more scalable SMT solvers by exploiting advances in SAT technology. The last few years have seen a great deal of interest and research on the foundational and practical aspects of SMT. SMT solvers have been developed in academia and industry with increasing scope and performance. SMT solvers or techniques have been integrated into: interactive theorem provers for high-order logic (such as HOL, Isabelle, and PVS), extended static checkers (such as Boogie and ESC/Java 2), verification systems (such as ACL2, Caduceus, SAL, UCLID, and Why), formal CASE environments (such as KeY), model checkers (such as BLAST, Eureka, MAGIC and SLAM), certifying compilers (such as Touchstone and TVOC), unit test generators (such as DART, EXE, CUTE and PEX).

This section provides a brief overview of SMT and its main approaches, together with references to the relevant literature for a deeper study. In particular, it focuses on the two most successful major approaches so far for implementing SMT solvers, usually referred to as the "eager" and the "lazy" approach.

2.5.1 . Background

Satisfiability Modulo Theories (SMT) problem is a decision problem for logical first order formulas with respect to combinations of background theories such

as: arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is a new and efficient SMT Solver freely available from Microsoft Research. It is used in various software verification and analysis applications.

In our study, we work in the context of (classical) first-order logic with equality and also use Z3. In order to better understand SMT, we introduce here the relevant basic concepts and notation. In the following, a theory T is a set of closed first-order formulas. A formula F is called $T - satisfiable$ or $T - consistent$ when $F \wedge T$ is satisfiable in the first-order sense. In the other case, it is $T - unsatisfiable$ or $T - inconsistent$. A partial assignment M will occasionally be seen as a conjunction of literals and also as a formula. If M is a $T - consistent$ partial assignment and F is a formula such that $M \models F$, i.e., M is a (propositional) model of F, then M is called a $T - model$ of F. If F and G are both formulas then F entails G in T, if $F \wedge \neg G$ is $T - inconsistent$. The short notation is $F \models_T G$. In the case that both $F \models_T G$ and $G \models_T F$ are true, then F and G are called $T - equivalent$. A clause C is called a theory lemma if $\varnothing \models_T C$ is true. Given a theory T and a formula F, one SMT problem is deciding whether F is $T - satisfiable$. For a background theory T, only the SMT problem for ground (and hence quantifier-free) CNF formulas F will be considered. These formulas often contain so-called free constants which are constant symbols not in the signature of T. When satisfiability is concerned, these constants can be considered as existential variables. Excluding the free constants, all other function and predicate symbols in the formulas will come from the signature of T. The only theories that will be considered are such that the $T-satisfiability$ of conjunctions of such ground literals is decidable. Any decision procedure for this problem is called a $T - solver$.

2.5.2 . SMT Algorithm

There are traditionally two approaches for deciding the satisfiability of a ground formula F with respect to a background theory T: the eager approach and the lazy approach.

Eager SMT techniques

The eager approach is based on devising efficient, specialized translations to convert an input formula into an equisatisfiable propositional formula using enough relevant consequences of the theory T. The approach applies in principle to any theory with a decidable ground satisfiability problem, possibly however at the cost of a significant blow-up in the translation. Its main allure is that the translation imposes upfront all theory-specific constraints on the SAT solver's search space, potentially solving the input formula quickly, in addition, the translated formula can be given to any off-the-shelf SAT solver. Its viability depends on the ability of

modern SAT solvers to quickly process relevant theory-specific information encoded into large SAT formulas. Eager encoding methods have been demonstrated for the following theories:

1. Equality and uninterpreted functions.
2. Integer linear arithmetic.
3. Restricted lambda expressions, such as arrays, memories.
4. Finite-precision bit-vector arithmetic.
5. Strings.

Using the eager algorithm makes easy to express SMT problem into SAT problem, but in this case one does without efficient specific algorithms that exist for solving corresponding theories, e.g., systems of linear equations: indeed, the SAT solver cannot use these algorithms after the SMT problem has been encoded into a SAT problem. Moreover, as the eager algorithm is not highly modular, the user has to design and code separately each theory, and thus, when different theories are mixed, it is difficult to ensure that the coding methods are compatible. The correctness of the eager technique for SAT depends on both the correctness of the translation, which is unique for each theory and the SAT theory. Despite the effort spent in making efficient translations, on many practical problems the SAT solver or the translation process run out of either time or memory. The techniques explained below are much faster.

Lazy SMT techniques

The lazy approach consists in building ad-hoc procedures implementing, in essence, an inference system specialized on a background theory T. The main advantage of theory-specific solvers is that one can use whatever specialized algorithms and data structures are best for the theory in question, which typically leads to better performance. The common practice is to write theory solvers just for conjunctions of literals, i.e., atomic formulas and their negations. These pared-down solvers are then embedded as separate sub-modules into an efficient SAT solver, allowing the joint system to accept quantifier-free formulas with an arbitrary Boolean structure. Given a formula F, each of its atoms that has to be checked is considered at first as a propositional symbol, $forgetting$ the theory T. Then the SAT solver will determine whether F is satisfiable, if not, F is $T - unsatisfiable$, otherwise the SAT solver will return a propositional model M of F and the assignment (which is only seen as a conjunction of literals) is checked by a $T - solver$. M is a $T - model$ of F if it is $T - consistent$. If not, the $T - solver$ builds a ground clause that is a logical consequence of T (a theory lemma), precluding that assignment. Then this lemma is added to F and the SAT solver is started again until the SAT solver returns $UNSAT$ or a $T - model$.

2.5.3 . SMT Solver

SMT-LIB

When using SMT solvers, it is important to follow the SMT-LIB [11] standard, which is characterised by:

1. Provide standard rigorous descriptions of background theories used in systems, i.e., SMT logics [11].

2. Develop and promote common input and output language for SMT solvers, i.e., SMT-LIB standard [13].

3. Build a SMT community to connect the SMT developers, researchers, and users.

4. Create and make available to the research community a large library of benchmarks.

5. Collect and promote software tools useful to the community. This is achieved through an annual competition called The Satisfiability Modulo Theories Competition or SMT-COMP [14].

The SMT-LIB standard [13] defines concepts, formal languages, and a command language. It also introduces the concepts of Theories and Logics in order to classify problems or instances. A problem belongs to a logic, a logic refers to some theories, and a theory is a specific set of symbols together with a set of axioms that defines a well-known system. Almost all SMT solvers support SMT-LIB, the standard used in the annual SMT competition.

SMT solver Z3

SMT solvers, or automatic theorem provers for SMT, are pieces of software designed as tools to automatically decide the satisfiability of a given formula related to a set of theories. Generally, a solver core relies on the DPLL(T) procedure. Z3 [68] is a microsoft research's SMT solver, which is one of the most advanced theorem provers to check the satisfiability of logic formulas on one or more background theories. Its development was targeted at solving problems that arise in software verification and software analysis. It has been considered as the overall most reliable solver by winning SMT-COMP from its beginning in 2007 until 2017 (since 2014, it has been the symbolic winner because its non-competitive participation). Z3 takes advantage of an open and strong design strategy which helps to drastically improve its performances.

Figure 2.4 depicts the schematic overview of Z3, which integrates a modern DPLL-based SAT solver, a *core theory solver* that handles equalities and uninter-

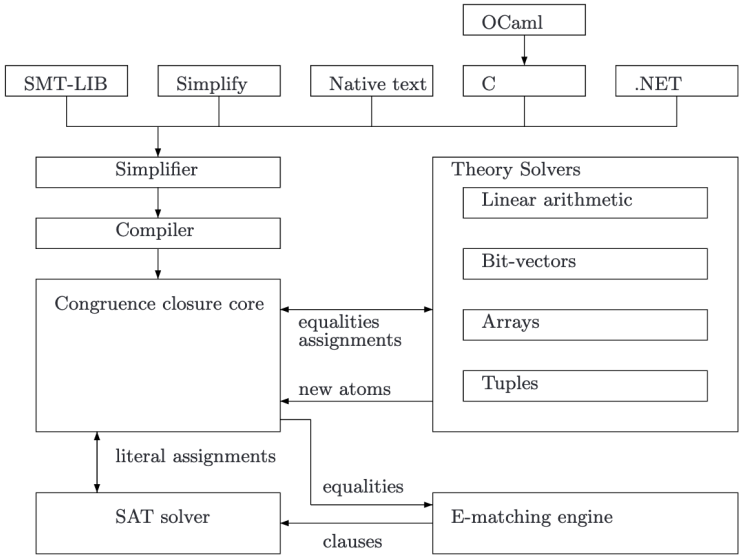

Figure 2.4: A schematic overview of Z3.

preted functions, *satellite solvers* (for arithmetic, arrays, etc.), and an *E-matching abstract machine* (for quantifiers).

As a low-level tool, Z3 is mainly regarded as a component applied to other tools that need to solve logical formulas. For ease of use, Z3 provides many APIs, and these APIs support languages such as C, .NET, and OCaml. At the same time, Z3 can also be executed directly through the command line. Its core is the simplifier and comparator, and its simplification strategy is very efficient. For example, it simplifies $p \wedge true$ to p, $x = 4 \wedge f(x)$ to $f(4)$, etc. In addition to software verification and program analysis, it is also used in other industrial applications for its powerful functions, such as in the field of network security, where Z3 can be used to solve problems such as cryptography, binary reversal, symbolic execution, etc.

2.5.4 . SMT-based Diagnosability Encoding for RTSs

Subsections 2.5.4, 2.5.5 and 2.5.6 refer to my work during my master internship [55].

Recall that the existence of timed critical pairs violates diagnosability of TA. However, the time duration for both trajectories should be infinitely long. For a real system that satisfies diagnosability defined in Definition 8 (2), i.e., for which such a timed critical pair does not exist, it is possible that a faulty trajectory may be distinguished from normal ones only after an unacceptably long time. To be more practical when applied to real systems, we use $\Delta \in R$ to represent a time upper bound after the fault occurrence to identify it, i.e., we rest on the concept

of Δ-diagnosability as defined in Definition 8 (1). So, saying that F is not Δ-diagnosable means that it exists a Δ-faulty run which cannot be distinguished by observation from at least one normal timed trajectory, i.e., it exists a Δ-critical pair as defined in Definition 9, whose existence violates Δ-diagnosability.

From Theorem 2, diagnosability (resp., timed critical pair) corresponds thus to $+\infty$-diagnosability (resp. $+\infty$-critical pair), if we extend Definition 7 to $\Delta = +\infty$ by replacing $> \Delta$ by $= +\infty$. Consider the case of an infinite Δ-CP. As time$(\rho, F) > \Delta$ (possibly $+\infty$), it exists a finite Δ-faulty prefix of ρ ending by a time transition. It results that we can replace, in the statement of Theorem 2, Δ-CP by finite Δ-CP ending by a time transition.

2.5.5 . Encoding Timed Automaton

In this section, we will show how to logically encode in SMT a Δ-CP for a TA such that, if the SMT solver finds a model for the proposed logic formula, then the considered fault is not Δ-diagnosable in this TA (and conversely if the formula length is large enough). In this case, the corresponding model, that is actually a Δ-CP, is finally returned. As we saw it is enough to look for a finite Δ-CP, it is thus possible to encode it by a finite formula. We will then consider bounded length timed trajectories with length parameter n, i.e., diagnosability checking is done on timed trajectories with length n, denoted by $L^n(A) = \{\rho \in L(A) \mid |\rho| = n\}$, where n varies from 1 to a fixed given bound. As explained in Section 2.2.2, we can assume that time and discrete transitions alternate in any timed trajectory. Hence, we rewrite $(q, v) \xrightarrow{t} (q, v'') \xrightarrow{\sigma} (q', v')$, where $t \in R$ and $\sigma \in \Sigma$, as $(q, v) \xrightarrow{t,\sigma} (q', v')$. In the following, we consider this kind of combined time-discrete transition during the encoding. Accordingly, a timed trajectory of length n is a finite sequence $(t_0, \sigma_0), (t_1, \sigma_1), ..., (t_{n-1}, \sigma_{n-1})$, where $t_i \in R$, $\sigma_i \in \Sigma$, and $\forall i,\ 0 \leqslant i \leqslant n-1,\ (q_i, v_i) \xrightarrow{t_i,\sigma_i} (q_{i+1}, v_{i+1})$ is allowed by A. As seen above, we can assume that the timed trajectory ends by a time transition, that we will represent by setting $\sigma_{n-1} = \epsilon$ as an unobservable event. For the example in Figure 2.3, one 4-length timed trajectory is $\rho = (1.5, o1), (3, u), (0.5, o2), (1, \epsilon)$ that is witnessed by the feasible execution $(q_0, x = 0) \xrightarrow{1.5,o1} (q_1, x = 0) \xrightarrow{3,u} (q_3, x = 3) \xrightarrow{0.5,o2} (q_5, x = 0) \xrightarrow{1,\epsilon} (q_5, x = 1)$. Given a TA $A = (Q, \Sigma, X, \delta^X, q_0, I)$, we encode essential static parts in A as follows:

- the set of states is encoded by positive integers through the function $E_Q : Q \to Q^E = \{1, ..., \|Q\|\}$.
- the set of events is encoded by positive integers $E_\Sigma : \Sigma \to \Sigma^E = \{1, ..., \|\Sigma\|\}$, where $\Sigma^E = \Sigma_o^E \uplus \Sigma_u^E \uplus \Sigma_f^E$, corresponding to $\Sigma = \Sigma_o \uplus \Sigma_u \uplus \Sigma_f$. We assume that normal events $\Sigma_n = \Sigma_o \uplus \Sigma_u$ are encoded by integers from 1 to $\|\Sigma_n\|$ and fault events by integers from $\|\Sigma_n\| + 1$ to $\|\Sigma\|$.

- the set of symbolic transitions is encoded by a set of tuples $E_{\delta^X} : \delta^X \rightarrow \delta^E \subseteq (Q^E \times \mathbb{C}(X) \times \Sigma^E \times 2^X \times Q^E)$ such that $E_{\delta^X}(q, g, \sigma, r, q') = (E_Q(q), g, E_\Sigma(\sigma), r, E_Q(q'))$.

2.5.6 . Encoding Bounded Diagnosability

In this section, given a TA with fault F and given Δ, duration after (first occurrence of) F, we show how to define, for arbitrary integers $k, \hat{k}$, a formula $\Psi_\Delta^{k,\hat{k}}$ whose satisfiability is equivalent to the existence of a Δ-CP $(\rho, \hat{\rho})$ with $|\rho| = k$ and $|\hat{\rho}| = \hat{k}$. In order to describe this formula as intuitively as possible, we present it with different separate parts. Since the satisfiability of $\Psi_\Delta^{k,\hat{k}}$ represents the existence of a Δ-CP, two timed trajectories ρ and $\hat{\rho}$ are concerned, which makes this property more complicated to encode (in absence of an explicit construction of the twin plant) than for example safety properties, for which it is not necessary to compare between different timed trajectories. To distinguish the value of variables between the two timed trajectories, the variables equipped with a hat are associated to the normal trajectory $\hat{\rho}$ of length $\hat{k}$ while the variables without a hat are attached to the faulty trajectory ρ of length k.

- The integer-valued variables $e_0, ..., e_{k-1}$ (resp. $\hat{e}_0, ..., \hat{e}_{\hat{k}-1}$) encode the events in the faulty (resp. normal) timed trajectory (with e_{k-1} and $\hat{e}_{\hat{k}-1}$ encoding ϵ).
- The integer-valued variables $s_0, ..., s_k$ (resp. $\hat{s}_0, ..., \hat{s}_{\hat{k}}$) represent the states in the faulty (resp. normal) timed trajectory.
- The real-valued variables $t_0, ..., t_{k-1}$ (resp. $\hat{t}_0, ..., \hat{t}_{\hat{k}-1}$) encode the time periods in the faulty (resp. normal) timed trajectory.
- The real-valued variables $v_0^x, ..., v_k^x$, $\forall x \in X$ (resp. $\hat{v}_0^x, ..., \hat{v}_{\hat{k}}^x$) represent the values of the corresponding clock x in each state in the faulty (resp. normal) timed trajectory, initialized as 0, i.e., $v_0^x = \hat{v}_0^x = 0$.
- The real-valued variables $v_0^t, ..., v_k^t$ (resp. $\hat{v}_0^t, ..., \hat{v}_{\hat{k}}^t$) encode the values of an additional global clock that should be initialized as 0 but never be reset to 0.
- The additional real-valued variables $v_0^F, ..., v_k^F$ represent the time elapsed after the first fault occurrence in the faulty timed trajectory (-1 by convention before the fault occurrence).

Initialization

The two timed trajectories should start in the initial state with the initialization of all clock variables.

- for the faulty timed trajectory:

$$\Phi^{Init} := (\bigwedge_{x \in X \cup \{t\}} v_0^x = 0) \wedge (v_0^F = -1) \wedge (s_0 = E_Q(q_0))$$

- for the normal timed trajectory:

$$\hat{\Phi}^{Init} := (\bigwedge_{x \in X \cup \{t\}} \hat{v}_0^x = 0) \wedge (\hat{s}_0 = E_Q(q_0))$$

Well-formedness of timed trajectories

The well-formedness of timed trajectories represents the fact that each time period between two discrete transitions should be nonnegative. Furthermore, the value of integer-valued variables representing all events in the two trajectories should be in $\{1...\|\Sigma\|\}$ for the faulty one and in $\{1...\|\Sigma_n\|\}$ for the normal one. This is encoded as follows.

- for the faulty timed trajectory:

$$\Phi^{WF} := (\bigwedge_{i=0}^{k-1} 0 \leqslant t_i) \wedge (\bigwedge_{i=0}^{k-1} 1 \leqslant e_i \wedge e_i \leqslant \|\Sigma\|) \wedge (\bigwedge_{i=0}^{k-1} 1 \leqslant s_i \wedge s_i \leqslant \|Q\|)$$

- for the normal timed trajectory:

$$\hat{\Phi}^{WF} := (\bigwedge_{i=0}^{\hat{k}-1} 0 \leqslant \hat{t}_i) \wedge (\bigwedge_{i=0}^{\hat{k}-1} 1 \leqslant \hat{e}_i \wedge \hat{e}_i \leqslant \|\Sigma_n\|) \wedge (\bigwedge_{i=0}^{\hat{k}-1} 1 \leqslant \hat{s}_i \wedge \hat{s}_i \leqslant \|Q\|)$$

Acceptance of timed trajectories

We formalize here that the two timed trajectories represented by values for the predefined variables as described above should be accepted by A. Precisely, in each timed trajectory, each pair of adjacent states has to be connected by a transition that is allowed in A.

- for the faulty timed trajectory:

$$\Phi^{Acc} := (\bigwedge_{i=0}^{k-1} (\bigvee_{(s_i, g, e_i, r, s_{i+1}) \in \delta^E} [[g]]_i \wedge \P_i^r))$$

  Here $[[g]]_i$ represents that the clock valuations after the i-th period in the faulty timed trajectory, i.e., $v_i^x + t_i$, should satisfy the guard g, such as:

  - $[[x \bowtie c]]_i := (v_i^x + t_i) \bowtie c$
  - $[[x - y \bowtie c]]_i := (v_i^x - v_i^y) \bowtie c$

– $[[g_1 \wedge g_2]]_i := [[g_1]]_i \wedge [[g_2]]_i$

$\P_i^r$ in the above expression formalizes the time progression, i.e., time transition, by resetting clocks in the subset r and by increasing all other clocks, including the global one (and also the time elapsed from the first fault occurrence if triggered) with the corresponding period t_i:

$$\P_i^r := (\bigwedge_{x \in r} v_{i+1}^x = 0) \wedge (\bigwedge_{x \in (X \setminus r) \cup \{t\}} v_{i+1}^x = v_i^x + t_i) \\ \wedge (0 \leqslant v_i^F \Rightarrow v_{i+1}^F = v_i^F + t_i)$$

- for the normal timed trajectory:

$$\hat{\Phi}^{Acc} := (\bigwedge_{i=0}^{\hat{k}-1} (\bigvee_{(\hat{s}_i, g, \hat{e}_i, r, \hat{s}_{i+1}) \in \delta^E} [[\hat{g}]]_i \wedge \hat{\P}_i^r))$$

In a similar way, $[[\hat{g}]]_i$ for the normal timed trajectory is encoded as follows:

– $[[\widehat{x \bowtie c}]]_i := (\hat{v}_i^x + \hat{t}_i) \bowtie c$

– $[[\widehat{x - y \bowtie c}]]_i := (\hat{v}_i^x - \hat{v}_i^y) \bowtie c$

– $[[\widehat{g_1 \wedge g_2}]]_i := [[\hat{g_1}]]_i \wedge [[\hat{g_2}]]_i$

The following is the time progression for the normal timed trajectory:

$$\hat{\P}_i^r := (\bigwedge_{x \in r} \hat{v}_{i+1}^x = 0) \wedge (\bigwedge_{x \in (X \setminus r) \cup \{t\}} \hat{v}_{i+1}^x = \hat{v}_i^x + \hat{t}_i)$$

Fault

The faulty timed trajectory contains a fault occurrence (with one fault type, the fault occurrence can be simplified as $\|\Sigma_n\| + 1 = e_i$)). Furthermore, after the first occurrence of a fault at step i, the value of the variable v_{i+1}^F is assigned to 0 to trigger counting the time elapsed from this fault occurrence (otherwise it stays equal to -1).

$$\Phi^F := (\bigvee_{i=0}^{k-1} \|\Sigma_n\| < e_i) \wedge (\bigwedge_{i=0}^{k-1} (v_i^F = -1 \Rightarrow \\ ((\|\Sigma_n\| < e_i \Rightarrow v_{i+1}^F = 0) \wedge (e_i \leqslant \|\Sigma_n\| \Rightarrow v_{i+1}^F = -1))))$$

Equivalent observations

The next condition to consider for a timed critical pair is that both timed trajectories should have exactly the same observations, i.e., the same observable event should be observed at the same time, which can be guaranteed by the global clock (that is never reset). This condition can be encoded as follows.

$$\Phi^{\equiv_{obs}} := (\bigwedge_{\sigma \in \Sigma_o^E} (\bigwedge_{i=0}^{k-1} (e_i = \sigma \Rightarrow (\bigvee_{j=0}^{\hat{k}-1} (\hat{e}_j = \sigma \wedge v_i^t = \hat{v}_j^t))))$$
$$\wedge (\bigwedge_{i=0}^{\hat{k}-1} (\hat{e}_i = \sigma \Rightarrow (\bigvee_{j=0}^{k-1} (e_j = \sigma \wedge v_j^t = \hat{v}_i^t)))))$$

For allowing null time elapse between two successive discrete transitions, i.e., allowing multiple simultaneous event occurrences, we require in addition (by means equality of ordered lists) that, at a same given time, all simultaneous observable events occur in the same order in both trajectories.

Time elapsed after fault

The last condition is that the time elapsed after the first occurrence of a fault is greater than Δ, which is encoded as follows.

$$\Phi_\Delta^{Time} := v_k^F > \Delta$$

Bounded diagnosability checking

We have formalized all conditions required for two timed trajectories to constitute a Δ-CP. Now we are ready to define:

$$\Psi_\Delta^{k,\hat{k}} := \Phi^{Init} \wedge \hat{\Phi}^{Init} \wedge \Phi^{WF} \wedge \hat{\Phi}^{WF}$$
$$\wedge \Phi^{Acc} \wedge \hat{\Phi}^{Acc} \wedge \Phi^{F} \wedge \Phi^{\equiv_{obs}} \wedge \Phi_\Delta^{Time}$$

Note that for the sake of simplicity, in the proposed formula, we do not handle state invariants. However, one can extend $\Psi_\Delta^{k,\hat{k}}$ by adding such constraints in a quite straightforward way. It suffices to enrich Φ^{Acc} and $\hat{\Phi}^{Acc}$ by verifying that the clock valuations in each state do not violate the corresponding invariant, which has to be done only when entering the state and leaving it.

Theorem 3 *Given a TA A and a considered fault F, F is Δ-diagnosable iff, for all $k, \hat{k}$, $\Psi_\Delta^{k,\hat{k}}$ is not satisfiable in A.*

2.6 . CEGAR and RECAR Algorithms

This section is partly borrowed from [63]. SAT technology has proven to be a very successful practical approach to solve some NP-complete problems. One of the main issues is to find the "right" encoding for the problem, i.e. to find a polynomial reduction from the original problem into a propositional formula in Conjunctive Normal Form (CNF, a set of clauses) which can be efficiently solved by a SAT solver. The SAT solver is a generic problem solving engine, whose input is a satisfiability equivalent CNF representing the original problem. It often happens

that either the SAT solver can solve efficiently the CNF or not at all (see, e.g., the results of the SAT competitions). A particular case is when the resulting CNF is very large: the time for generating and reading the input is greater than the time to solve it. This is due to the limited available main memory allocated to the approach and not the SAT solver itself.

For huge CNF encodings, specific approaches have been designed in the past, where a SAT solver is used as an oracle in a more complex procedure. One such procedure is called CounterExample-Guided Abstraction Refinement (CEGAR) [36]. The SAT solver is fed with an abstraction of the original problem allowing more models (which we will call under-approximation). If the abstraction is unsatisfiable, then the original problem is also unsatisfiable (UNSAT shortcut). Else the procedure is able to verify if the model found for that abstraction is a correct solution for the original problem. In this case, we have an additional SAT shortcut to decide the satisfiability of the formula. If it is not the case, new constraints are added to prevent the solver from finding such spurious examples (refinement step) and the process repeats. Eventually, a complete satisfiability equivalent propositional formula is provided, and the SAT solver can decide the problem. One of the reasons for using CEGAR is that the complete formula is in practice too large to even be generated, so the only hope to solve the original problem is to "get lucky" (satisfiable shortcut) or to be able to take into account a specific structure of the problem (unsatisfiable shortcut).

This framework is elegant and has been applied to many areas: Satisfiability Modulo Theory, Planning and more recently QBF. The latter is especially inspiring, because it appears to be the best practical solution overall to solve QBF formulas according to the latest QBF competition. The aim of the RECAR extension [63] was to follow these steps on another PSPACE complete problem, which is the satisfiability of modal logic K formulas.

2.6.1 . CEGAR Algorithm

CounterExample-Guided Abstraction Refinement (CEGAR) is an incremental way to decide the satisfiability of formulas in classical propositional logic. It has been originally designed for model checking [36], i.e., to answer questions such as "Does $S \models P$ hold?" or, equivalently, "Is $S \wedge \neg P$ unsatisfiable?", where S describes a system and P a property. In such highly structured problems, it is often the case that only a small part of the formula is needed to answer the question. The idea behind CEGAR is to replace $\phi = S \wedge \neg P$ by an approximation ϕ', where ϕ' is easier to solve in practice than ϕ. There are two kinds of approximations:

- (1) An over-approximation of ϕ is a formula $\hat{\phi}$ such that $\hat{\phi} \models \phi$ holds: $\hat{\phi}$ has at most as many models as ϕ;
- (2) An under-approximation of ϕ is a formula $\check{\phi}$ such that $\phi \models \check{\phi}$ holds: $\check{\phi}$

has at least as many models as ϕ;

where ϕ is usually a CNF.

Over-approximation

The over-approximation has less models than the original problem, it receives a problem ϕ as input and outputs a problem $\hat{\phi}$ which is more constrained than ϕ. Thus if $\hat{\phi}$ has a model, then so does ϕ. A classical way to over-approximate is to bound the generation of the formula ϕ to a given n smaller than the one needed to reach equi-satisfiability to the original problem (as in bounded model checking or planning). As such a model of $\hat{\phi}$ can be extended to a model of ϕ but the unsatisfiability of $\hat{\phi}$ means that the bound n has to be increased and the process is repeated. Such over-approximation is successfully used in planning by bounding the size of the authorized plan [76].

Under-approximation

Similarly, the under-approximation has more models than the original problem, it also receives a problem ϕ as input and outputs a problem $\check{\phi}$ which is less constrained than ϕ. Thus if $\check{\phi}$ has no model, then so does ϕ. A classical way to under-approximate ϕ is to "forget" some clauses, i.e., ϕ is a subset of the clauses in ϕ. A model of $\check{\phi}$ also may by chance satisfy ϕ. This double possibility to conclude earlier makes under-approximation based CEGAR very popular. Such under-approximation is successfully used in constraint-based reasoning under the name of *Relaxation* [40].

So, we have two different types of approximations and different ways to implement them. Over-approximation and under-approximation are the basis of CEGAR, which is a framework to make "user-friendly" the use of these approximations to solve problems. CEGAR usually takes into consideration decision problems and is of two kinds, according to which kind of approximation is adopted.

CEGAR with over-approximation

CEGAR-over represents the CEGAR framework instantiated with over-approximation. An example of CEGAR using over-approximations is given in Figure 2.5. It receives a formula ϕ as input and computes an over-approximation ψ of ϕ. Then it uses a SAT solver to check whether ψ is satisfiable. If so it concludes that ϕ is satisfiable. Otherwise, ψ is refined, i.e., it gets closer to ϕ, until it is satisfiable, or until the refined over-approximation is detected to be equi-satisfiable to ϕ, denoted $\psi \equiv_{\mathsf{sat}} \phi$, where it concludes that ϕ is unsatisfiable. In the following, $\phi \equiv^{?}_{\mathsf{sat}} \psi$ means an incomplete efficient equi-satisfiability test which returns yes or unknown. Recent SAT solvers are able to check satisfiability "under assumption" [45], i.e.,

given the satisfiability of a set of literals called assumptions, and to provide in case of unsatisfiability a "reason" in terms of those literals for the unsatisfiability of a formula.

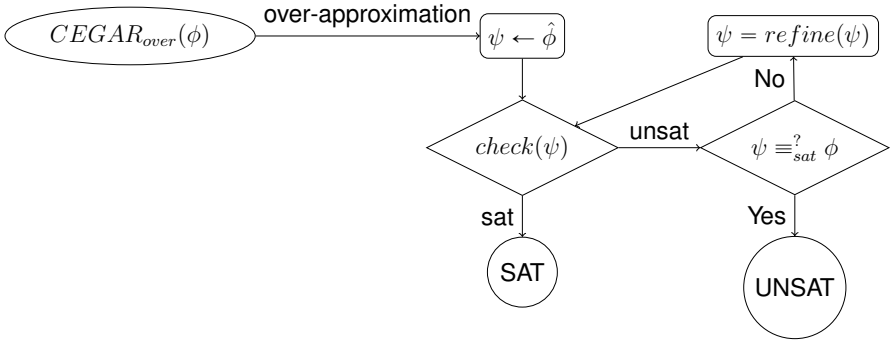

Figure 2.5: The CEGAR framework with over-approximation.

CEGAR with under-approximation

CEGAR-under represents the CEGAR framework instantiated with under-approximation. An example of CEGAR using under-approximations is given in Figure 2.6. It receives a formula ϕ as input and computes an under-approximation χ of ϕ. Then it calls SAT oracle to check whether χ is unsatisfiable. If so it can conclude that ϕ is unsatisfiable. Otherwise, χ is refined closer to ϕ, until it is unsatisfiable, or until the model λ for the refined under-approximation is detected to be also a model for ϕ, denoted $\lambda \models^{?} \phi$, in this case it concludes that ϕ is satisfiable.

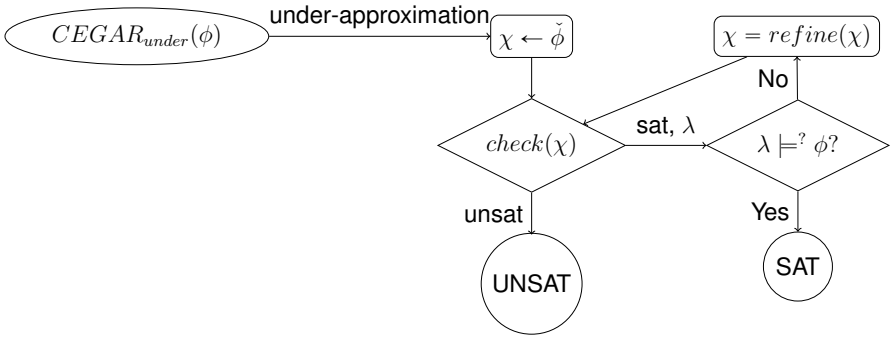

Figure 2.6: The CEGAR framework with under-approximation.

2.6.2 . RECAR Algorithm

A classic CEGAR approach with over-approximation and a SAT shortcut performs well when the input is satisfiable. But generally, it does not perform well in problems which are unsatisfiable. The reason is that it has then to keep refining until it reaches equi-satisfiability with the original problem. Conversely a CEGAR approach with under-approximation and UNSAT shortcut does not in general perform well in problems which are satisfiable. One way to address this issue is to mix SAT and UNSAT shortcuts, as in [29] and [97]. In these approaches, the methods alternate between over and under approximations.

The Recursive Explore and Check Abstraction Refinement (RECAR) approach [63] is a sound, complete and terminating framework to mix both over- and under-approximation and to be able to interleave them during the search. As in the CEGAR framework, it can be used in two modes. We present the one where the first call is on an over-approximation.

RECAR-over

The RECAR-over approach, depicted in Figures 2.7 and Algorithm 2.8, interleaves both kinds of approximation each abstraction is performed with the information retrieved from solving the previous one. The UNSAT shortcut is implemented using a recursive call to the main procedure when a strict under-approximation $\check{\phi}$ can be built. One should also note that the proposed approach permits abstractions on two different levels: one is used to simplify the problem at the domain level (recursive call), while the other one is used to approximate the problem at the oracle level. In order to apply RECAR, the under-approximation $\check{\phi}$ and the over-approximation $\hat{\phi}$ must satisfy some properties. In the following, isSAT(ϕ) means that ϕ is satisfiable ($\not\models_1 \neg\phi$) and isUNSAT (ϕ) means ϕ is unsatisfiable ($\models_2 \neg\phi$), but on possibly different consequence relations. $RC(\phi, \check{\phi})$ denotes a Boolean function deciding if a Recursive Call should occur. The RECAR-over assumptions are thus:

1. Function 'check' is a sound and complete implementation of 'isSAT' which terminates.
2. isSAT ($\hat{\phi}$) implies isSAT(refine ($\hat{\phi}$)).
3. There exists $n \in \mathbb{N}$ such that refine $^n(\hat{\phi}) \equiv^?_{\text{sat}} \phi$.
4. isUNSAT ($\check{\phi}$) implies isUNSAT (ϕ).
5. Let $\text{under}(\phi) = \check{\phi}$. There exists $n \in \mathbb{N}$ such that RC (under $^n(\phi)$, under $^{n+1}(\phi)$) evaluates to false.

Note that we have isSAT ($\hat{\phi}$) implies ϕ is satisfiable by Assumptions 2 and 3 together.

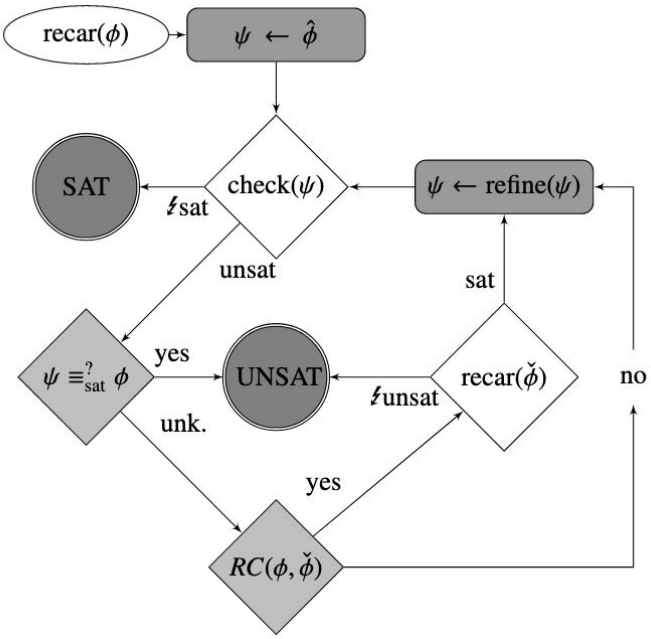

Figure 2.7: The RECAR framework

Under these assumptions, [63] proves, based on the presentation of the algorithm *recar*(ϕ) in Figure 2.8, that RECAR is sound (if *recar*(ϕ) returns SAT then ϕ is satisfiable), complete (if *recar*(ϕ) returns UNSAT then *isUNSAT*(ϕ)) and terminates (RECAR terminates for any input ϕ).

Function 1: recar(ϕ)

```
1 ψ ← φ̂
2 while ψ ≡?sat φ returns "unknown" do
3     if check(ψ) = SAT then return SAT ;
4     if RC(φ, φ̌) then
5         if recar(φ̌) = UNSAT then return UNSAT ;
6     ψ ← refine(ψ) ;
7 return check(ψ)
```

Figure 2.8: The RECAR algorithm

This RECAR-over framework has successfully been used to solve modal logic K satisfiability problem [63].

3 - An Approximation-based Incremental SMT-based Approach to Diagnosability Analysis of Real-Time Systems

In this chapter, we focus on improving the efficiency of diagnosability checking for real-time systems modeled as timed automata. Inspired by an extension of the classical CEGAR (CounterExample-Guided Abstraction Refinement) algorithm introduced recently, namely the RECAR (Recursive Explore and Check Abstraction Refinement) algorithm, we propose new RECAR-like algorithms that combine over- and under-approximation techniques. We use CEGAR for early termination of over- and under-approximation refinement loop, in the case the original formula is satisifiable or unsatisfiable respectively, and then we show soundness of our RECAR-like approach applied to an arbitrary formula. Finally, we evaluate the effectiveness of our method on different benchmarks using the SMT solver Z3 by comparing with the naïve method without approximation shortcut.

3.1 . Motivation

In our previous research, we found that when verifying the diagnosability on an automaton (finite state automaton or timed automaton), a considerable part of the transitions in the state space is not relevant, i.e., it will not affect the result, which provides the possibility to improve the efficiency. In consequence, we take advantage of the new paradigm of resolution of RECAR to verify diagnosability of timed systems with a good efficiency in practice.

Work on diagnosability has mainly focused on models of discrete event systems and real-time systems. It has been shown that any pair of sufficiently long trajectories sharing the same observation, one of which is faulty and the other is normal (called (timed) critical pair), constitutes a witness of non-diagnosability.

From previous research, it was proved that diagnosability verification problem of discrete event systems was in the class P for finite state automata and became PSPACE-complete for timed automata. Combining SAT and SMT technologies have proved to be a very successful practical approach to solve some NP-complete problems, achieving good results in this case. Nevertheless, scalability is not guaranteed and too large automata cannot be handled. As RECAR appears promising for dealing with beyond NP problems, in particular PSPACE-complete problems, we verify diagnosability of real-time systems by using the RECAR framework.

CEGAR (see section 2.6.1 for detail) is an incremental way to decide the satisfiability of formulas in classical propositional logic, based on the use of either over- or under-approximations of the original formula. We saw that CEGAR-over does not perform well for unsatisfiable problems whereas CEGAR-under does not

perform well for satisfiable problems and that one way to address this issue is to alternate between over- and under-approximations. The RECAR approach (see section 2.6.2 for detail) precisely interleaves both kinds of approximations.

In order to verify the bounded Δ-diagnosability of real-time systems with the RECAR framework, we propose a RECAR-like approach, which includes refinement iteration steps that may switch between CEGAR-over and CEGAR-under by estimating the ease of verifying a next refinement or of using another type of approximation. In each iteration step, CEGAR starts by refining the formula which had been the last one processed by CEGAR of the same type.

This chapter is organized as follows. Section 3.2 introduces CEGAR with over-approximations and refinement determined by three changeable parameters for diagnosability checking problem. Section 3.3 introduces CEGAR with under-approximations and refinement determined by two changeable parameters. We also define the *semi-equivalence* of observations instead of equivalence as an under-approximation for an easier check of diagnosability. Finally, we consider reducing the number of formulas to satisfy to perform an under-approximation at the oracle level. Section 3.4 proposes a RECAR-like algorithm alternating CEGAR-over and CEGAR-under, by defining a switch function SF to decide whether to switch from one type of approximation to the other one. Section 3.5 shows how to encode changeable parameters in SMT in order to support the incremental processing of refinements. Section 3.6 presents the construction of a scalable benchmark and experimental results of the CEGAR and RECAR-like algorithms on several of its instances. Section 3.7 analyzes the experiment results, indicates that the key factor of complexity and thus non-efficiency is the length of the critical pair searched, which cannot be improved with our under-approximations for UNSAT problems. Section 3.8 concludes and draws perspectives of future research.

3.2 . CEGAR-over for Bounded Diagnosability Analysis of RTS

As we stated above, CEGAR is a framework to make easily use of over- or under-approximations to solve problems. It is widely used in a large number of decision problems. There are two kinds of CEGAR according to which kind of approximation function we are using. In order to remain sound and complete, CEGAR techniques are typically combined with a refinement loop.

Figure 3.1 shows the CEGAR-over framework for diagnosability checking problem. It receives original formula ϕ as input, representing a parameterized critical pair candidate, and over-approximates it to ψ at first, then calls SMT solver to check whether ψ is satisfiable. If so we can conclude ϕ is satisfiable, i.e., the existence of parameterized critical pairs. Otherwise, ψ is refined to make it closer to ϕ, until it is satisfiable, or until the refined over-approximation is equal to ϕ ,

denoted $\psi = \phi$, in which case we can conclude that ϕ is unsatisfiable.

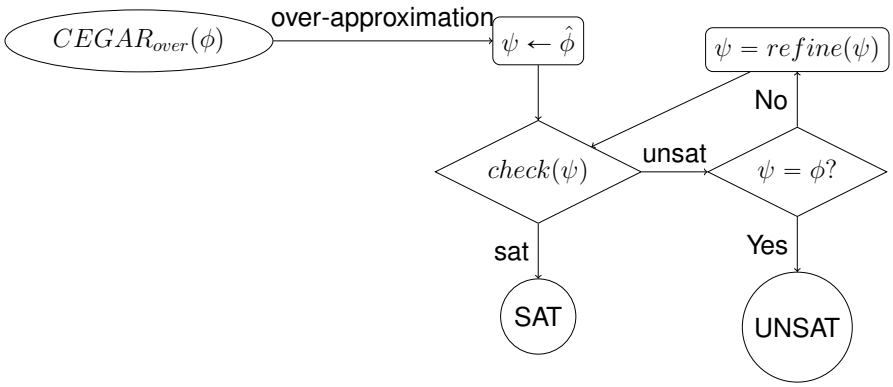

Figure 3.1: The CEGAR-over framework for diagnosability checking.

In the classical CEGAR-over way depicted in Figure 2.5, it usually defines an equisatisfiable function for ψ and ϕ to terminate the refinement process, denoted $\psi \equiv_{sat} \phi$. In our study, we have not defined an equisatisfiable function between ϕ and ψ for over-approximation, so here we simply set $\psi = \phi$. Figure 3.2 illustrates that the successive over-approximations are becoming closer and closer to the original problem ($\subseteq$ meaning entailment of formulas or inclusion of their models, and $refine^i(\hat{\phi})$ representing the i^{th} refinement on $\hat{\phi}$).

$$\phi \xrightarrow{over-approximation} \hat{\phi} \subseteq refine^1(\hat{\phi}) \subseteq refine^2(\hat{\phi}) \subseteq \blacksquare\,\blacksquare\,\blacksquare \subseteq refine^n(\hat{\phi}) = \phi$$

Figure 3.2: Over-approximation refinement process.

Let $A = (Q, \Sigma, X, \delta^X, q_0, I)$ be a timed automaton as explained in Definition 4, where:

- $\delta^X \subseteq Q \times \mathbb{C}(X) \times \Sigma \times 2^X \times Q$ is a finite set of transitions (q, g, σ, r, q'), where the guard $g \in \mathbb{C}(X)$, which has to be satisfied for the transition to be fired, and the clocks $r \subseteq X$ reset to zero, when not specified, are by default $true$ and $\varnothing$, respectively;

- Σ is the set of events as the labels of δ^X, which we assume partitioned as $\Sigma = \Sigma_o \uplus \Sigma_u \uplus \Sigma_f$.

- $I : Q \to \mathbb{C}(X)$ is the invariant function that associates with each state q the invariant $I(q)$, a constraint that has to be satisfied by clocks in state q, whose value by default, when not specified, is $true$. We require $\mathbf{0} \in [\![I(q_0)]\!]$.

For given bound B and Δ, the diagnosability verification for TA consists in checking the existence of a Δ-critical pair (see Definition 9) of length at most B. I.e., a pair of trajectories issued from q_0, the length of each one at most B, composed of labeled transitions in δ^X, one trajectory being a Δ-faulty run (see Definition 7), i.e., containing a transition labeled by the fault F, with a period after the first occurrence of F equal to Δ, and the other trajectory being correct, while sharing the same timed sequence of observable events. In other words, the existence of a timed critical pair violates the Δ-diagnosability of TA.

Now, we can represent the diagnosability problem as $P = CP(B, \Delta, \delta^X, \Sigma_o)$, where B and Δ are given by the user, and δ^X is the set of transitions that constitute the system (timed automaton), i.e., the search space of the SMT solver, and Σ_o the observable events, all of these parameters being changeable in the various approximations during the diagnosability verification with the CEGAR algorithm. In order to apply the CEGAR framework, we denote the over- or under-approximation of the original problem P as $P' = CP(B', \Delta', \delta^{X'}, \Sigma'_o)$.

We define three types of over-approximations of the bounded Δ-diagnosability problem according to three parameters:

- $Bound$ (parameter B). Decrease the bound (length) admissible, denoted by $P' = CP(B', \Delta, \delta^X, \Sigma_o)$, where $B' < B$. So if the SMT oracle returns SAT, we prove *non-Δ-diagnosability*, if UNSAT we can refine this over-approximation by refining B' to B'' with $B' < B'' < B$.

- $Transition\ set$ (parameter δ^X). Decrease the set of transitions used, denoted by $P' = CP(B, \Delta, \delta^{X'}, \Sigma)$, where $\delta^{X'} \subset \delta^X$. So if the SMT oracle returns SAT, we prove *non-Δ-diagnosability*, if UNSAT we can refine this over-approximation by refining $\delta^{X'}$ to $\delta^{X''}$ with $\delta^{X'} \subset \delta^{X''} \subset \delta^X$.

- $Observable\ event\ set$ (parameter Σ_o). Increase the set of observable events, denoted by $P' = CP(B, \Delta, \delta^X, \Sigma'_o)$, where $\Sigma' = \Sigma'_o \uplus \Sigma'_u \uplus \Sigma_f$ and $\Sigma'_o \supset \Sigma_o$ (thus $\Sigma'_u = \Sigma_u \backslash (\Sigma'_o \backslash \Sigma_o)$), i.e., turning some unobservable events into observable ones. So if the SMT oracle returns SAT, we prove *non-Δ-diagnosability*, if UNSAT we may refine this over-approximation by refining Σ'_o to Σ''_o with $\Sigma'_o \supset \Sigma''_o \supset \Sigma_o$ (thus $\Sigma''_u = \Sigma'_u \cup (\Sigma'_o \backslash \Sigma''_o)$).

Obviously all three parameters can be combined (with one to three simultaneous changes) for defining an over-approximation, denoted by $P' = CP(B', \Delta, \delta^{X'}, \Sigma'_o)$. Thus the refinement is defined by increasing the bound (while keeping it smaller than the initial one) and/or increasing the subset of admissible transitions and/or decreasing the set of observable events (while keeping it greater than the initial one).

3.3 . CEGAR-under for Bounded Diagnosability Analysis of RTS

Indeed, the CEGAR framework can also be instantiated with under-approximation.

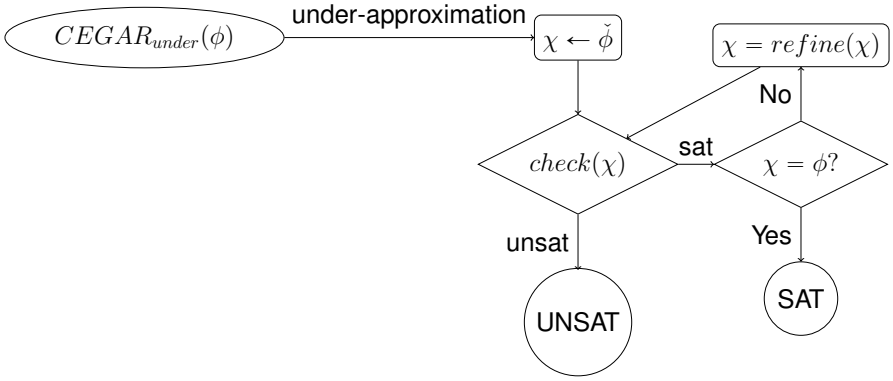

Figure 3.3: The CEGAR-under framework for diagnosability checking.

CEGAR-under for diagnosability checking problem is given in Figure 3.3. It receives original formula ϕ as input and under approximates it to χ, then calls SMT solver to check whether χ is unsatisfiable. If so we can conclude that ϕ is unsatisfiable, i.e., the non-existence of parameterized critical pairs. Otherwise, χ is refined to make it closer to ϕ, until it is unsatisfiable, or until the refined under-approximation is equal to ϕ, denoted $\chi = \phi$, in which case we can conclude that ϕ is satisfiable.

In the classical CEGAR-under framework depicted in Figure 2.6, the refinement process possibly early terminates when the model λ for under-approximation χ is detected to be also adapted to ϕ, denoted $\lambda \models^{?} \phi$. In this case, ϕ is satisfiable. In our study, since we have not checked whether the model for χ could be extended in a model for ϕ, similar to CEGAR-over, we simply set $\chi = \phi$ here. Figure 3.4 illustrates that the successive under-approximations are becoming closer and closer to the original problem.

$$\phi \xrightarrow{under-approximation} \check{\phi} \supseteq refine^1(\check{\phi}) \supseteq refine^2(\check{\phi}) \supseteq \blacksquare\blacksquare\blacksquare \supseteq refine^n(\check{\phi}) = \phi$$

Figure 3.4: Under-approximation refinement process.

As with CEGAR-over, we have defined two types of under-approximations according to two parameters:

- $Time\ after\ fault\ occurrence$ (parameter Δ). Decrease Δ, denoted by $P' = CP(B, \Delta', \delta^X, \Sigma_o)$, where $\Delta' < \Delta$. So if the SMT oracle returns

UNSAT, we prove Δ-*diagnosability*. As non-existence of Δ'-critical pair implies non-existence of Δ-critical pair. Otherwise, if the oracle returns SAT, we may refine this under-approximation by Δ'' with $\Delta' < \Delta'' < \Delta$.

- $Observable\ event\ set$ (parameter Σ_o). Decrease the set of observable events, denoted by $P' = CP(B, \Delta, \delta^X, \Sigma'_o)$, where $\Sigma' = \Sigma'_o \uplus \Sigma'_u \uplus \Sigma_f$ and $\Sigma'_o \subset \Sigma_o$ (thus $\Sigma'_u = \Sigma_u \cup (\Sigma_o \backslash \Sigma'_o)$), i.e., turning some observable events into unobservable ones. So if the SMT oracle returns UNSAT, we prove Δ-*diagnosability*. As non-existence of Δ-critical pair with less observable events guarantees non-existence with more observable events. If SAT we may refine this under-approximation by refining Σ'_o to Σ''_o with $\Sigma'_o \subset \Sigma''_o \subset \Sigma_o$ (thus $\Sigma''_u = \Sigma'_u \backslash (\Sigma''_o \backslash \Sigma'_o)$).

Obviously both parameters can be combined (with one to two simultaneous changes) for defining an under-approximation, denoted by $P' = CP(B, \Delta', \delta^X, \Sigma'_o)$. Thus the refinement is defined by increasing the time after fault occurrence (Δ) (while keeping it smaller than the initial one) and/or increasing the set of observable events (while keeping it smaller than the initial one). Note that the parameter Σ_o is the only one that can be used both for over- and under-approximation, depending on its increase or decrease, respectively.

Now, we can also define an under-approximation by weakening the condition of equality of timed observations between the faulty and the normal trajectories of the timed critical pair.

Definition 10 (Semi-Equivalence of Observations) *Given a TA A, the considered fault F, two timed trajectories $\rho, \rho' \in L(A)$, with ρ faulty (containing F) and ρ' normal (not containing F,) are observationally semi-equivalent, if $P(\rho) \subseteq P(\rho')$.*

where P is the projection of the timed trajectory to observable events . Obviously semi-equivalence is weaker than equivalence as only inclusion of timed observations is required instead of equality. For an easier diagnosability checking, we look for a pair of timed trajectories such that the observations of the faulty one are included in the observations of the normal one, regardless of whether the two trajectories can form a critical pair. If SMT oracle returns UNSAT, we can conclude that ϕ is unsatisfiable. We could as well define semi-equivalence in the other way, i.e., observations of the normal trajectory included in those of the faulty trajectory, but this requirement is trivially satisfied by taking as normal trajectory any prefix of the faulty trajectory before the first fault occurrence.

Typically, based on SMT, a real-time system and its diagnosability property are represented by formulas. These formulas are combined into a single verification formula which is then checked by an SMT solver. The solver tries to refute the diagnosability property by finding a counter-example where the system violates this property. If the formula is satisfiable, most modern SMT solvers can generate a model of it. This model can be used to construct a concrete execution of the timed

critical pair that leads to the diagnosability violation. In order to apply the RECAR framework, we use the over- and under-approximations of the initial formula defined above. For under-approximations we can also, similarly to the classical approach, simply "forget" some part of the diagnosability formula in conjunctive form, i.e., select a partial diagnosability formula for verification, which effectively reduces the size of formulas.

3.4 . A RECAR-like Approach for Bounded Diagnosability Analysis of RTS

CEGAR-over is efficient for early determination, through over-approximation refinement loops, of satisfiability of an original formula which is satisfiable. CEGAR-under is efficient for early determination, through under-approximation refinement loops, of unsatisfiability of an original formula which is unsatisfiable. However, CEGAR-over is not performing well for an original formula which is unsatisfiable and CEGAR-under is not performing well for an original foumula which is satisfiable. For a system, in order to verify its diagnosability efficiently, inspired by the RECAR algorithm, we present a new framework that we call RECAR-like approach, which is sound, complete and terminating. It alternates between CEGAR-over and CEGAR-under during search of a solution, i.e., it switches the type of approximation of CEGAR when it estimates that this will be more efficient than continuing to refine the same type. In the following, $isSAT(\phi)$ means that ϕ is satisfiable and $isUNSAT(\phi)$ means ϕ is unsatisfiable.

Definition 11 (RECAR-like Assumptions) *To be able to perform a RECAR-like framework to decide a problem, one must verify a list of assumptions. We assume that $isSAT$ is the satisfiability in the corresponding logic of $\hat{\phi}$, and $isUNSAT$ is the unsatisfiability in the corresponding logic of $\check{\phi}$.*

1. *Function $check$ is a sound and complete implementation of $isSAT$ (so of $isUNSAT$ also) which terminates.*
2. $isSAT(\hat{\phi})$ *implies* $isSAT(refine(\hat{\phi}))$.
3. $isUNSAT(\check{\phi})$ *implies* $isUNSAT(refine(\check{\phi}))$.
4. *There exists* $n \in N$ *such that* $refine^n(\hat{\phi}) = \phi$ *and* $refine^n(\check{\phi}) = \phi$.
5. $isUNSAT(\check{\phi})$ *implies* $isUNSAT(\phi)$.
6. $isSAT(\hat{\phi})$ *implies* $isSAT(\phi)$.

Actually, 5 is a consequence of 3 and 4 and 6 is a consequence of 2 and 4. Assumptions 2 and 3 are satisfied by the way we have constructed refinements. The reason why assumption 4 is satisfied comes from the fact that most of the

parameters (B, δ^X, Σ_o) whose variations define the successive refinements have a finite domain of values. Only Δ parameter is continuous but we chose to make evolve it by integer multiples of a given time unit, so its domain of values is also finite.

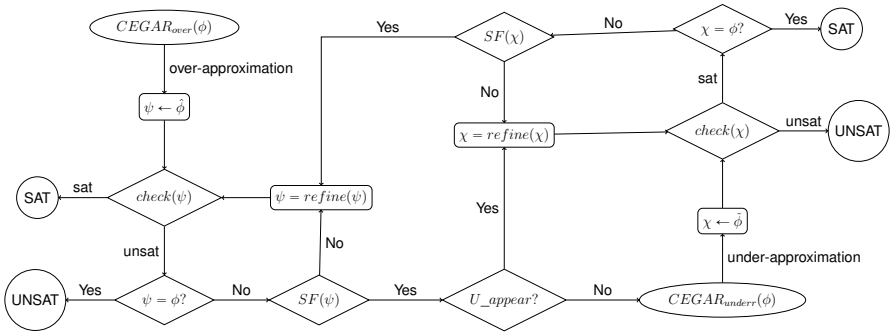

Figure 3.5: The RECAR-like framework for diagnosability checking.

Figure 3.5 shows how over-approximation and under-approximation techniques can be combined to solve diagnosability problem with a SMT solver. The switch function SF is a Boolean function that determines if an alternation should occur, which reflects an estimate of the efficiency of either continuing the refinement process or switching CEGAR type. Here, SF makes a conclusion by comparing the time of the last two checks. Thus the idea of RECAR-like approach is to alternate both kinds of approximations, and use switch function SF to decide whether to switch from CEGAR-over to CEGAR-under or conversely, otherwise to continue refining the over- or under-approximation until the result is obtained. U_appear is also a Boolean function that judges whether the under-approximation has already been called.

First of all, we perform CEGAR-over combination of the three parameters. Then, if we cannot make any conclusions, we decide whether to move to CEGAR-under with a combination of its two parameters (plus possibly using semi-equivalence). In each iteration we call the SMT solver. Depending on the result we have to perform an additional function to decide whether to continue with the RECAR-like algorithm, i.e. to judge if the approximation reaches the original formula.

For the CEGAR-over, if the result is unsatisfiable we need to check if the current formula is equivalent to the original one. If not, we have to decide whether moving to CEGAR-under. As each alternation starts by refining the formula which had been the last one processed by CEGAR of the same type, we also need to check whether CEGAR-under has been called before entering CEGAR-under.

3.5 . Encoding RECAR-like Approach

In this section, we will show how to encode our RECAR-like approach combining python and SMT solver Z3. First, we encode TA and *bounded Δ-diagnosability* as explained in Section 2.5.5 and Section 2.5.6. As the CEGAR algorithm consists in an iteration process, we use selectors to choose the values of the changeable parameters in SMT in order to support the incremental processing of the refinements.

3.5.1 . Pre-processing

Before encoding CEGAR and RECAR, in order to improve efficiency, the data is pre-processed and the transitions are sorted as follows:

1. We define and pre-compute an $index$ value to each transition, which represents the shortest distance from the initial state to the source state of this transition. In this way, in the search process, for a specific bound value, the SMT solver Z3 only searches for satisfiable transitions in the space consisting of those transitions whose $index$ is not greater than $bound$, so the search space of Z3 is greatly reduced.

2. The fault diagnoser (Definition 21) and the normal diagnoser (Definition 22) are pre-computed and their transitions indexed (a same transition may belong to both diagnosers, so have both indexes), so that Z3 searches the transitions for the faulty path only in the faulty diagnoser and for the normal path only in the normal diagnoser.

In our previous work (see subsection 2.5.6 for detail), we have theoretically proved the correctness and feasibility of our algorithm for verifying diagnosability of real-time systems. By adopting item 1, the optimization effect becomes more evident with larger state spaces. Note that we executed all the experiments by keeping the same experimental conditions: running on a Mac OS laptop with a 2.7 GHz Intel Core i5 processor and 8 Go 1600 MHz DDR3 of memory. Similarly, we performed the same experiments on two benchmarks.

Tables 3.1 and 3.2 show part of our experimental results before and after optimization respectively, where the 2nd column shows the upper bound B for the length of the critical pair and the time Δ after the fault occurrence chosen, the 3rd shows the size of the formula expressed by its number of clauses, the 4th is the required memory, the 5th in 3.1 presents the number of transitions of the system and in 3.2 presents the number of transitions whose index is not greater than B, the 6th gives the execution time in seconds (including in 3.2 the preprocessing time for computing the transitions whose index is not greater than B) and the last one is the satisfiability verdict.

	B/Δ	$\|clauses\|$	$mem.$	$\|trans.\|$	$time$	$SAT?$
$hvac_1$	9/6	345756	33	15	26	Yes
$hvac_2$	5/3	496963	69	15	30	No
$hvac_3$	9/5	465393	52	15	21	Yes
$hvac_4$	6/3	737395	63	15	39	No
$hvac_5$	15/7	1353321	75	156	152	Yes
$hvac_6$	7/6	1709069	129	156	306	No
ex_1	21/15	246563	21	123	21	Yes
ex_2	9/6	903296	285	123	135	No

Table 3.1: Experimental results before optimization

	B/Δ	$\|clauses\|$	$mem.$	$\|trans.\|$ $within$ B	$time$	$SAT?$
$hvac_1$	9/6	345756	33	15	26	Yes
$hvac_2$	5/3	172836	16	11	13	No
$hvac_3$	9/5	465393	52	15	21	Yes
$hvac_4$	6/3	362738	43	12	16	No
$hvac_5$	15/7	701523	39	30	69	Yes
$hvac_6$	7/6	253648	31	16	22	No
ex_1	21/15	174536	17	28	16	Yes
ex_2	9/6	146328	14	16	12	No

Table 3.2: Experimental results after optimization

From Table 3.2, one can see that our optimization approach is feasible since even the hand-crafted versions terminated well within the timeout (that we set to 3600 seconds).

The experiments results show the advantage of the optimization approach on reducing the search space. Comparing with our experimental results shown in Table 3.1 corresponding to Section 2.5.6, one can see from Table 3.2 that the code efficiency has been greatly improved. Thus this optimization will be applied in all experiments that follow.

3.5.2 . Encoding Changeable Parameters

In the CEGAR approach using the over- and under-approximation, we play with the parameters $Bound$, $Time\ after\ fault\ occurrence$, $Transition\ set$, $Observable\ event\ set$, the values of all of which being able to be changed in the different refinement loops. We will call the corresponding variables L, D, T, O respectively, initialized to their respective values B, Δ, δ^X, Σ_o in the initial problem. We show in the following how to choose their values in SMT.

$Bound\ L$. Given $\{n_i\}$ the possible values of L in the over-approximation of the initial problem and its successive refinements (e.g., all integers from 1 to B),

we use a formula that associates a fresh variable l_i (a selector) to each value n_i. In the over-approximation and its refinements, L is thus set by the SMT solver to the given value n_i by setting l_i to $True$ (then the l_j's, $j \neq i$, are automatically set to $False$, as L has only one value). This is encoded as:

$$(L = n_i) \wedge l_i, n_i \in \mathbb{N}.$$

$Time\ after\ fault\ occurrence\ D$. Similar to bound, given $\{t_i\}$ a finite set of possible values of D in the under-approximation of the initial problem and its successive refinements (e.g., all integer multiples of a given fraction of Δ), we use a formula that associates a fresh variable d_i (a selector) to each value t_i. In the under-approximation and its refinements, D is set by the SMT solver to the given value t_i by setting d_i to $True$ and the d_j's, $j \neq i$, to $False$. This is encoded as:

$$(D = t_i) \wedge d_i, t_i \in R.$$

$Transition\ set\ T$. We associate a fresh variable (selector) tr_i to each transition $trans_i$ of the system. In the over-approximation and its successive refinements, T is set by the SMT solver to a given subset of δ^X by setting tr_i to $True$ for all those transitions $trans_i$ that belong to this subset and tr_j to $False$ for all those transitions $trans_j$ that do not belong to this subset. Taking into account that only those transitions whose pre-computed index is not greater than the present value of the bound L, the formula is finally encoded as:

$$\bigwedge_{trans_i \in \delta^X} ((trans_i \in T) \wedge (Index(trans_i) \leqslant L) \wedge tr_i).$$

.

$Observable\ event\ set\ O$. To make observable events changeable, we associate a fresh variable (selector) o_i to each event $e_i \in \Sigma \backslash \Sigma_f$. O is set by the SMT solver to a given subset of $\Sigma \backslash \Sigma_f$ (that contains Σ_o in the case of an over-approximation and its successive refinements, or is included in Σ_o in the case of an under-approximation and its successive refinements) by setting o_i to $True$ for all those events e_i that belong to this subset and o_j to $False$ for all those events e_j that do not belong to this subset. This is encoded as:

$$\bigwedge_{e_i \in \Sigma \backslash \Sigma_f} ((e_i \in O) \wedge o_i).$$

.

3.6 . Experiments

The correctness of our algorithm is a consequence of it for that is satisfied the RECAR-like assumptions of Defination 11. We have theoretically proved the correctness of our algorithm. To show its feasibility, we compared our RECAR-like approach with CEGAR-over and CEGAR-under with different parameters, and

carried out a prototype implementation done in Python by using the SMT solver Z3.

We reported on several versions of four benchmarks from literature. Considering that such literature examples are normally quite small, in order to show the scalability we tested hand-crafted versions of them where the state space was generated in a partially random way while keeping the verdicts. Furthermore, all of which have been modified in order to get diagnosable and non-diagnosable versions. The first literature benchmark is inspired by the HVAC system from [82], in which we added different temporal constraints since it was a finite automaton and an arbitrary number of events in the system in a way such that the verdict remains the same by enlarging the size of the TA. $hvac_2$ is a version modified by adding a clock to constraint the $close\_valve$ such that this delay is always different for a normal trajectory and for a faulty one. The system is diagnosable (No SAT). Then by modifying this delay on faulty trajectories without any constraint, the system becomes not diagnosable, as shown by example $hvac_1$. Similarly, the second group examples, denoted by $jiang_i$, are obtained by modifying the example from [59]. The third group examples from [104] and the fourth group examples are models of traffic light controller at a pedestrian crossing from [64], these two group examples are TA in the literature, we only modified them into two non-diagnosable and diagnosable versions and enlarging them to ensure scaliability, denoted by $zbrz_i$ and lee_i respectively. The last group examples, denoted by ex_i, are totally our hand-crafted ones for testing structurally complex systems, i.e., TA with many cycles.

	B	Δ	$\|trans.\|$	$Over\_L$	$Over\_T$	$Over\_O$	$Over\_all$	$Refinements$	$WithoutCEGAR$	$Switch$	$RECAR-like$	$SAT?$
$hvac_1$	50	50	500	0.40	3.23	8.33	0.35	4	6.57	0	0.36	Yes
$hvac_2$	50	50	500	2.40	5.15	10.12	2.47	50	4.4	0	2.51	No
$jiang_1$	50	50	500	0.10	2.32	5.33	0.09	3	5.15	0	0.11	Yes
$jiang_2$	50	50	500	2.00	4.23	8.32	2.05	50	1.77	0	2.11	No
$zbrz_1$	30	30	500	0.04	3.87	3.32	0.04	2	1.55	0	0.04	Yes
$zbrz_2$	30	30	500	0.80	5.48	8.96	0.62	30	1.56	0	0.65	No
lee_1	23	23	450	0.08	98.32	110.22	0.06	3	1.08	0	0.06	Yes
lee_2	23	23	450	>500	200.03	>500	>500	23	141.12	5	165.87	No
ex_1	20	16	34	0.66	3.96	4.98	0.48	10	3.85	0	0.46	Yes
ex_2	20	16	34	>500	189.32	>500	>500	20	217.97	3	264.35	No

Table 3.3: Experimental results for CEGAR-over

Table 3.3 shows part of our experimental results for CEGAR-over with different over-approximations, where the 2nd and 3rd column show the $Bound\ B$ and $Time\ after\ fault\ occurrence\ \Delta$ given as parameters of the initial problem, the 4th column shows the number of transitions of the system, the 5th, 6th, 7th columns give the execution times in seconds of CEGAR-over for approximations defined by the $Bound\ L$, $Transition\ set\ T$, $Observable\ event\ set\ O$ respectively, the 8th and 9 th columns give the execution times and the number of refinements of CEGAR-over with a combination of these three parameters, the 10th and 11th columns give the execution times of the direct checking of diagnosability without using CEGAR and using our RECAR-like approach, the last column in the table

provides the diagnosability verdict. Execution times for CEGAR-over techniques are obtained by adding the execution time for the first over-approximation chosen and those for each successive refinement of this approximation until the verdict is obtained.

	B	Δ	$\|trans.\|$	$Under\_D$	$Under\_O$	$Under\_all$	$Refinements$	$WithoutCEGAR$	$Switch$	$RECAR-like$	$SAT?$
$hvac_1$	50	50	500	25.65	38.33	22.31	50	6.57	0	0.36	Yes
$hvac_2$	50	50	500	7.94	7.45	6.95	3	4.4	0	2.51	No
$jiang_1$	50	50	500	30.12	28.33	24.22	50	5.15	0	0.11	Yes
$jiang_2$	50	50	500	2.01	3.13	5.32	3	1.77	0	2.11	No
$zbrz_1$	30	30	500	4.04	6.15	3.88	30	1.55	0	0.04	Yes
$zbrz_2$	30	30	500	3.50	3.33	3.32	7	1.56	0	0.65	No
lee_1	23	23	450	3.20	4.32	3.44	23	1.08	0	0.04	Yes
lee_2	23	23	450	455.77	>500	482.28	4	141.12	5	165.87	No
ex_1	20	16	34	15.56	9.78	10.92	20	3.85	0	0.46	Yes
ex_2	20	16	34	344.32	202.17	210.06	8	217.97	3	264.35	No

Table 3.4: Experimental results for CEGAR-under

Table 3.4 shows part of our experimental results for CEGAR-under with different under-approximations. The first 4 columns and last 2 columns are the same as Table 3.3. The 5th, 6th columns give the execution times of CEGAR-under for approximations defined by the $Time\ after\ fault\ occurrence\ D$ and $Observable\ events\ set\ O$ respectively, the 7th column shows the execution times for the approximation defined by checking $semi-equivalence$ of observations instead of equivalence, the 8th and 9th columns give the execution times and the number of refinements of CEGAR-under with a combination of the two parameters D and O. Execution times for CEGAR-under techniques are obtained by adding the execution time for the first under-approximation chosen and those for each successive refinement of this approximation until the verdict is obtained.

Refinement Strategies

We explain below the refinement strategies used, i.e., how the initial values of the parameters defining the first over- or under-approximation are chosen and how their changes are fixed in the successive refinements.

- $Bound\ L$. In CEGAR-over, L is initialized to 1, i.e., $n_1 = 1$, and then at each refinement step the value of L is increased by 1, i.e. $n_{i+1} = n_i + 1$, until we obtain the satisfiable conclusion, or until $n_i = B$ is satisfied, in which case if the SMT solver still returns unsat, we obtain the unsatisfiable conclusion. So, there are at most B refinements. In CEGAR-under, L is fixed to B ($L = B$) and there is no refinement.

- $Time\ after\ fault\ occurrence\ D$. In CEGAR-under, D is initialized to one-tenth of the initial Δ, i.e., $t_1 = 0.1 \times \Delta$ and then at each refinement step the value of D is increased by one-tenth of Δ, i.e., $t_{i+1} = t_i + 0.1 \times \Delta$, until we obtain the unsatisfiable conclusion, or until $t_i = \Delta$ is satisfied, in which case if the SMT solver still returns sat, we obtain the satisfiable conclusion.

In this way, we can ensure that refinement occurs at most 10 times. In CEGAR-under, D is fixed to Δ ($D = \Delta$) and there is no refinement.

- $Transition\ set\ T$. In CEGAR-over, T is initialized to some nonempty subset of those transitions in δ^X based on B, i.e., whose index is not greater than B (e.g., 10% of them, randomly chosen) and then at each refinement step, others among those transitions (e.g., another randomly chosen 10% of them) are added to T, until we obtain the satisfiable conclusion or until T equals the whole set of transitions based on B, in which case if the SMT solver still returns unsat, we obtain the unsatisfiable conclusion. If parameters T and L are mixed together in the definition of the over-approximation and its refinements, T is initialized to some nonempty subset of those transitions in δ^X based on n_1 (the initial value of L), and then at each refinement step some nonempty subset of new transitions based on n_i (the current value of L in this refinement) are added to T. In CEGAR-under, T is fixed to the subset of δ^X based on B and there is no refinement.

- $Observable\ event\ set\ O$. In CEGAR-over, O is initialized to $\Sigma \backslash \Sigma_f$, i.e, treating all unobservable events as observable ones, with the exception of faults, and then, at each refinement step, we remove an unobservable event (the first remaining according to a random order on $\Sigma_u \backslash \Sigma_f$) from O, until we obtain the satisfiable conclusion or until $O = \Sigma_o$, in which case if the SMT solver still returns unsat, we obtain the unsatisfiable conclusion. In CEGAR-under, O is initialized to a single observable event (the first according to a random order on Σ_o), then in each refinement step we add another observable event to O (the first remaining according to the previous order), until we obtain the unsatisfiable result or until $O = \Sigma_o$, in which case if the SMT solver still returns sat, we obtain the satisfiable conclusion.

As usual our bounded model checking approach can verify non-diagnosability, i.e., SAT cases, as bounded non-diagnosability implies non-diagnosability. However, note that bounded diagnosability, i.e., UNSAT cases, does not imply diagnosability, as the problem could become SAT when increasing the given initial bound B.

3.7 . Results and Discussion

The experimental results in Tables 3.3 and 3.4 show, as expected, that CEGAR-over techniques with over-approximations parameterized by the parameter $Bound\ L$ or the three mixed parameters $Bound\ L$, $Transition\ set\ T$, $Observable\ event\ set\ O$ speed up checking of satisfiable instances, meanwhile slow down checking of most unsatisfiable instances. Unfortunately, CEGAR-under techniques with under-approximations are inefficient not only (as expected) for satisfiable instances but also (which was not expected) for unsatisfiable instances. This is the key factor

that will cause inefficiency also in RECAR-like approaches. Based on these visual data, we summarize further observations and analyze them.

First, CEGAR-over with parameter $Bound\ L$ performs as good as CEGAR-over with mixed three parameters $Bound\ L$, $Transition\ set\ T$, $Observable\ event\ set\ O$ on the satisfiable instances. However, CEGAR-over with only parameter $Transition\ set\ T$ or parameter $Observable\ event\ set\ O$ performs worse on the same instances. Second, CEGAR-over performs poorly on most unsatisfiable benchmarks. Third, CEGAR-under does not perform well on all benchmarks, even on the instances that are unsatisfiable. It results that our RECAR-like approach is more efficient to verify diagnosability for satisfiable instances but shows no improvement for unsatisfiable instances. The reasons for this situation are as follows:

1. For unsatisfiable instances, a CEGAR approach using an over-approximation and its refinements defined earlier will take a long time before finally conclude as the refinement will have to reach the initial problem, i.e., $\hat{\phi} = \phi$ (no SAT shortcut).

2. For satisfiable instances, a CEGAR approach using and under-approximation and its refinements defined earlier will take a long time before finally conclude as the refinement will have to reach the initial problem, i.e., $\check{\phi} = \phi$ (no UNSAT shortcut).

3. The key factor of complexity and thus non-efficiency seems to be the length of the critical pair searched, thus the size of the formula checked for satisfiability. For over-approximations, this length L is one parameter that can be considerably reduced and can lead to a SAT shortcut for satisfiable instances. But for under-approximations, this length has to be kept to the value B given for the $Bound$, thus bringing no improvement even for unsatisfiable instances. This very probably explains why CEGAR-under, and thus RECAR-like approach, remains inefficient for unsatisfiable instances.

With the RECAR-like approach, we begin by using the CEGAR-over algorithm to check the system diagnosability. If this is not conclusive, then there is the chance to switch to CEGAR-under when the switch function SF is triggered. If the instance is satisfiable, since the switch function SF is evaluated based on the time difference between the current and the last verification times, there is little chance of triggering the SF function because over-approximations with parameter $Bound$ L always perform well for satisfiable instances even after several refinement loops, i.e., the last verification time usually is almost no longer than the previous one, due also to the use of incremental SMT. In this case, the RECAR-like approach keeps running the CEGAR-over algorithm, which is obviously good for the satisfiable cases, and effectiveness of the first comes from effectiveness of the second. If now the instance is unsatisfiable, we get the following situations:

1. The switch function SF is not triggered, i.e., it is always the CEGAR-over algorithm that is running and does not switch to CEGAR-under. This happens for example in the three first groups of examples in Table 3.3. CEGAR-over with L parameter or with combined L, T, O parameters is actually often more efficient than solving directly the problem even when the instances are unsatifiable, so is RECAR-like approach.

2. SF is triggered, then the CEGAR-under algorithm is executed, and, possibly after several switches, we get the unsatisfiable conclusion from CEGAR-under that returns unsat, in which case the RECAR-like algorithm contains the entire CEGAR-under process (which we saw is inefficient) plus part of the CEGAR-over process.

3. SF is triggered several times, similar to 2., but we get the unsatisfiable conclusion from CEGAR-over, so only after the condition $\hat{\phi} = \phi$ is satisfied, in which case the RECAR-like algorithm contains the entire CEGAR-over process plus part of the additional CEGAR-under process (which is inefficient).

Both cases 2 and 3 above contain the full or partial CEGAR-under process. As CEGAR-under does not improve efficiency because it does not play with the key factor $Bound\ L$, RECAR-like approach is inefficient for the unsatisfiable, i.e., diagnosable, instances.

3.8 . Conclusion

We proposed in this chapter a new approach for improving the efficiency of diagnosability problem using an alternating abstraction refinement approach. Meanwhile, we discussed different techniques and showed how they can be implemented on this problem enabling further optimizations like early sat termination. Finally, we evaluated the effectiveness of our approach on benchmarks from the literature using the SMT solver Z3. Our RECAR-like approach, mixing SAT and UNSAT shortcuts, has good performance for satisfiable (non-diagnosable) instances. From the fact it does not perform well for unsatisfiable (diagnosable) instances, we got the conclusion that the key factor of complexity is the length of the critical pair that is looked for. In future work, we will focus on how to play with this $Bound\ L$ parameter in under-approximations, in order to increase the efficiency of CEGAR-under and thus of our RECAR-like approach. We will also try to define under-approximations by "forgetting" some part of the diagnosability formula in conjunctive form, but preliminary experiments seem to show that, in the unsat case, most subformulas we tried are found to be satisfiable and an unsatisfable subformula would probably be close in size from the global formula. Another possible way is to play with time constraints expressed in the guards and invariants, i.e., tightening these constraints for defining over-approximations and relaxing them for defining under-approximations. But again very preliminary experiments seem to

show that for under-approximations the parameter L fixed to $Bound\ B$ remains the cause of inefficiency.

4 - Designing Diagnosable Discrete Event Systems by using Delay Blocks

We propose in this chapter a new non-intrusive way to make a non-diagnosable discrete event system diagnosable by merely adding delay blocks on some observable events. It consists in calculating the set of observable event occurrences whose deferral can turn a non-diagnosable system into a diagnosable. This calculation uses the max-flow min-cut theorem [48], which is encoded in SMT. We first present our motivations and the potential interests intended from studying such problem. Then we remind what finite automata with delay blocks (ADB) are. And we adapt the concept of ADB to eliminate all pairs of trajectories witnessing non-diagnosability by taking care not to create new ones, based on max-flow min-cut theorem. As far as we know, this is the first attempt to remove non-diagnosability with delay blocks without using controllable events, or changing the structure of systems. Our approach is encoded into an SMT formula, whose correctness and efficiency are then demonstrated by experimental results.

4.1 . Motivation/Introduction

We have reminded in the second chapter of this thesis the problem of diagnosability checking for discrete event systems (DESs). In a given DES, the existence of two infinite behaviors, with the same observations but one containing the considered fault and not the other, violates diagnosability. Existing work searches for such ambiguous behaviors both in centralized ([82, 59, 75, 35, 50]) and distributed ([73, 85, 100]) ways. The most classical method is to construct a structure called twin plant that captures all pairs of observationally equivalent behaviors to directly check the existence of such ambiguous pairs.

As diagnosability is one critical property of the system, it is important to design a diagnosable system, which will prove to be substantially convenient for subsequent system diagnosis. But this goal is not trivial and up to now, most work has focused on checking diagnosability of the system, while little work has considered improving diagnosability efficiently and economically. It is thus interesting to study how to transform non-diagnosable systems into diagnosable ones.

In this chapter, we propose a new non-intrusive approach for this purpose by merely deferring some observable events, while keeping the original system structure. For the sake of simplicity, this work has been conducted on classical DESs, modeled by finite state automata (FSA).

We calculate the set of occurrences of observable events whose deferral can make a non-diagnosable system diagnosable. To this end, we use the max-flow min-cut theorem, which is then encoded in SMT. The reason that we chose SMT instead

of SAT is to make our approach extensible in order to handle timed automata in a straightforward way in a later work. More precisely, we adapt finite automata with delay blocks (ADB) ([34]) to eliminate all pairs of trajectories witnessing non-diagnosability by taking care not to create new ones, based on the max-flow min-cut theorem, encoded in SMT, so that every faulty trajectory can be distinguished by observation from all normal ones.

Our contribution to the design of diagnosable DESs is multifold. First, in order to eliminate efficiently all pairs violating diagnosability by adding the fewest delay blocks possible, we calculate the minimum number of transitions in normal trajectories according to the max-flow min-cut theorem, encoded in SMT. Secondly, we analyze the scope of our approach by characterizing the systems for which it is applicable. Thirdly, we present experimental results on benchmarks to demonstrate the efficiency and correctness of our approach.

4.2 . Designing Diagnosable Systems with Delay Blocks

In this section, we will remind the concept of Automaton with Delay Blocks (ADB) and present how to use it to design a diagnosable system.

Given a discrete event system (DES) G, we model it as a FSA as explained in Section 2.2.1 (see Definition 2). Also, for the sake of simplicity, we adopt the Assumptions 1 and 2 about liveness of the system and the possibility to infinitely observe the infinite trajectories.

Now, based on Definition 5, diagnosability checking consists in verifying the non-existence of a critical pair (see Definition 6), which has been proven to violate diagnosability defined by Definition 5 and thus witnesses non-diagnosability. One classical algorithm to check this property is to construct a structure, often called twin plant in literature, obtained by synchronizing normal and faulty trajectories based on observable events such that we can directly check the existence of such a critical pair on the twin plant. More precisely, the twin plant is obtained as the product, synchronized on observable events, of the diagnoser of the system by itself. We will consider directly the refined form of the twin plant, obtained as the synchronized product (actually, in this case, just the product) of the refined diagnoser by itself.

Based on the Definition 20, each state of the diagnoser D_G is a pair made up of a system state and a fault label equal to F when a fault occurs on the path being considered from the initial state to the present state, N otherwise.

Example 4 *Figure 4.1 shows the diagnoser D_G of the system G depicted in Figure 2.2.*

The refined diagnoser is obtained by keeping only observable information, which is defined as the delay closure (see Definition 17) of the diagnoser w.r.t. the set of observable event Σ_o and denoted by $D_G^R = \mathbb{C}_{\Sigma_o}(D_G)$.

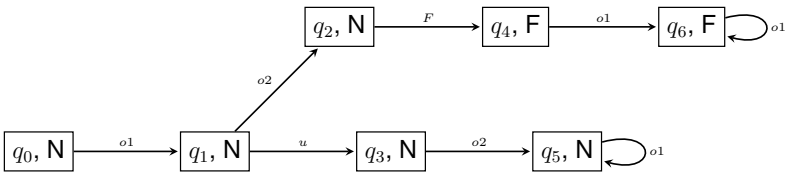

Figure 4.1: Diagnoser of the system in Figure 2.2

Example 5 *Figure 4.2 shows the refined diagnoser D_G^R constructed from the diagnoser D_G depicted in Figure 4.1.*

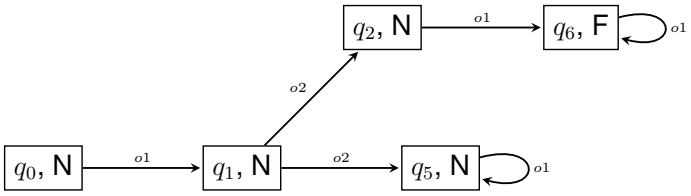

Figure 4.2: Refined diagnoser of the system in Figure 2.2

Definition 12 *(Twin plant). Given a system model G, its (refined) twin plant T_G is obtained by taking the product of the corresponding refined diagnoser D_G^R by itself, i.e., $T_G = D_G^R \parallel_{\Sigma_o} D_G^R = D_G^R \times D_G^R$.*

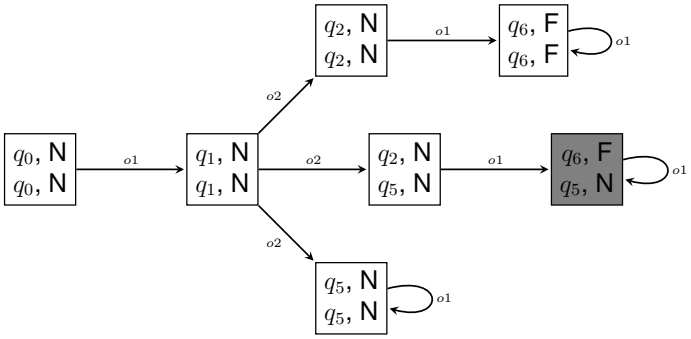

Figure 4.3: Twin plant of the system in Figure 2.2

Each state of a twin plant is composed of a pair of diagnoser states, which is called ambiguous state if one of the fault labels is F and the other is N. An ambiguous cycle is a cycle that contains only ambiguous states. Figure 4.3 depicts the twin plant based on the refined diagnoser shown in Figure 4.2. Note that the gray node is an ambiguous state, whose self-cycle is thus an ambiguous cycle.

As our goal is to identify all critical pairs in order to eliminate them later by adding delay blocks, we can further reduce the twin plant by retaining only its parts that are useful for this purpose.

Definition 13 *(Critical twin plant). Given a twin plant T_G, its critical version T_G^C is obtained by keeping only all its states from which an ambiguous cycle is reachable (and removing possible normal cycles that could remain on paths from the initial state to such an ambiguous cycle).*

Example 6 *The critical twin plant T_G^C obtained from the twin plant T_G of Figure 4.3 is shown in Figure 4.4.*

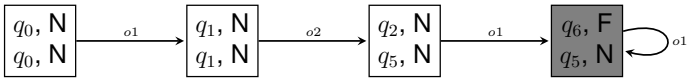

Figure 4.4: Critical twin plant of the system in Figure 2.2

4.2.1 . Automata with Delay Blocks (ADB)

We now introduce finite automata with delay blocks (ADB) ([34]) by adapting them to our problem. ADB, obtained by extending finite state automata with time delay blocks, can be used to defer some observable events. Our idea is to use the deferral aspect of ADB to eliminate all critical pairs in the twin plant without creating new ones, so that every faulty trajectory can be distinguished from normal ones.

Definition 14 *(ADB) An automaton with delay blocks (ADB) is a quintuple $\mathcal{B} = (Q, D, \Sigma, \delta, q_0)$, where:*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$\Sigma = \Sigma_o \uplus \Sigma_u \uplus \Sigma_f$ is a finite set of events;*
- *D is a finite set of delay blocks. Each delay block $d \in D$ is indexed by a natural number $t \geqslant 0$ to indicate the amount of delay for the outputs. We will limit here to two delay blocks: one indexed by 0 (no delay), the other by 1 (delay, say of one time unit);*
- *δ is a set of transitions, defined by*

$$\delta : (Q \times \Sigma_o \times D) \uplus (Q \times (\Sigma_u \uplus \Sigma_f)) \rightarrow 2^Q.$$

General delay blocks indexed by integers would be useful for a future extension of our work but, in the present untimed framework, we just need blocks with no delay or some non-null (fixed) delay, say 1, given that events will keep their usual order in each class (not deferred and deferred by 1). Note also that we do not need *tick* transitions.

Compared to the underlying FSA, i.e., the corresponding version without D, the order of observable events may be changed due to delay blocks. There are two kinds of transitions with different types of events:

- $\delta(q, \sigma, d) = Q'$, where $\sigma \in \Sigma_o$ and $Q' \subseteq Q$, denoting the transitions from q to a state in Q', with the event σ whose observation is either deferred when the associated value of d is 1, or not otherwise.

- $\delta(q, \sigma) = Q'$, where $\sigma \in \Sigma_u \uplus \Sigma_f$ and $Q' \subseteq Q$, unobserved transitions without delay blocks.

For a finite word w, let $|w|$ denote its length, and $w[i]$ its $(i+1)$th element (starting from 0), if $|w| > i$. Given an ADB $\mathcal{B}$, its timed language $L(\mathcal{B})$ is defined by:

$$L(\mathcal{B}) = \{w \in (\Sigma \times \{0,1\})^* \mid \exists q \in Q, (q_0, w, q) \in \delta\}$$

where the timed word w is a finite string of tuples $\langle \sigma, d \rangle \in \Sigma \times \{0,1\} = (\Sigma_o \times \{0,1\}) \uplus ((\Sigma_u \uplus \Sigma_f) \times \{0\})$, all unobservable events implicitly associated with 0. We refer to the first component of the tuple $w[i]$ as the $event$ and to the second as the $timestamp$, the latter denoting being deferred or not.

We redefine for an ADB the projection operator P of a finite timed trajectory w on observable events. Given w, $P(w) = P_0(w)1P_1(w)$ is made up of the concatenation of the two subsequences of observable events in w whose timestamps are all 0 or 1, respectively, with a delay of one between. Thus, given w and w', $P(w) = P(w')$ iff $P_0(w) = P_0(w')$ and $P_1(w) = P_1(w')$.

Consider the example in Figure 4.5, which is an ADB extended from the FSA of Figure 2.2. Here, the delay values for all observable transitions are 0 and omitted, except for the transition (q_3, o_2, q_5), where the value is 1. Consider $w_n = \langle o1, 0 \rangle \langle u, 0 \rangle \langle o2, 1 \rangle \langle o1, 0 \rangle^n$, then $P_0(w_n) = o1^{n+1}$ and $P_1(w_n) = o2$. Hence, F is now diagnosable in this ADB as the fault trajectory $w_{nF} = \langle o1, 0 \rangle \langle o2, 0 \rangle \langle F, 0 \rangle \langle o1, 0 \rangle^n$ is distinguished from normal ones by observations, as $P(w_n) = o1^{n+1}o2 \neq P(w_{nF}) = o1o2o1^n$ (here the order is enough to distinguish and we even do not need to know when the delay of one occurs but this is not true in general).

Take care that the observation language $P(L(B))$ is no longer live (as $o1^{n+1}o2$ is not the prefix of a longer word), but this does not matter as this language is not the one generated by the behavior of B but reflects just how it is observation is modified by the delays. Note also that the time of the delay (one unit) plays

actually no role, except differing the corresponding observable event at the end of the observation sequence. More than a time delay, it is actually a logical delay. But time delays would be necessary if we would consider time automata instead of automata.

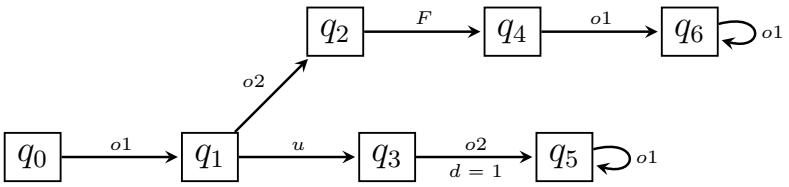

Figure 4.5: An example of ADB

Our novel diagnosable transformation approach, based on ADBs, comprises the following major steps:

1. Synchronizing on observable events the refined diagnoser D_G^R with itself to get the twin plant T_G.
2. Unfolding its critical version T_G^C into a flow network FN.
3. Calculating a constrained min-cut set of this flow network through SMT and adding to each selected transition of this min-cut a 1-valued delay block such that the corresponding critical pairs disappear in the resulting ADB.

4.2.2 . Unfolding FSA into Flow Network

We now briefly introduce the classical notion of flow network before showing how to unfold a given FSA into a specific flow network adapted to our approach (see [2] for more details about flow network theory).

Definition 15 *(Flow network) A flow network is a directed and connected graph* $FN = (V, E)$ *with a capacity function* w *assigning a weight to each edge:*

- V *is the set of nodes (vertices);*
- $E \subseteq V \times V$ *is the set of edges;*
- w *is a function :* $E \rightarrow \mathbb{R}_+$*;*
- *source node* $\iota \in V$ *and target node* $t \in V$ *are two distinguished nodes, with no incoming (resp., outgoing) edge of positive capacity for* ι *(resp.,* t*).*

Given an FSA, we construct its corresponding flow network. The idea is to adapt Floyd's cycle detection algorithm in order to link the last node of each cycle

to a newly created node, target node t (unfolding). We just extend slightly the definition of a flow network by keeping the original transition labels of the FSA in addition to the weights. And, in our case, each edge has the unique weight 1 since our goal is to find the minimal set of observable transitions that can make all critical pairs disappear by using additional delay blocks.

We thus unfold T_G^C as a flow network FN. The way we construct FN guarantees that any reachable cycle in T_G^C is transformed into a path reaching the target node t (note that T_G^C contains only ambiguous cycles). In other words, if we can block all paths from ι to t, we can then forbid all ambiguous cycles, thus all critical pairs. We will show, in the next section, how to use the max-flow min-cut theorem to calculate a minimal set of transitions from a given critical twin plant, that can block all its existing ambiguous cycles.

Example 7 *Figure 4.6 shows the unfolding FN of the critical twin plant T_G^C of Figure 4.4.*

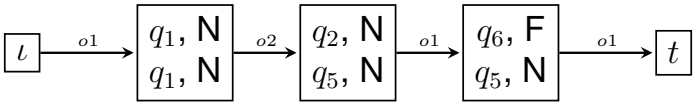

Figure 4.6: Unfolding the critical twin plant of Figure 4.4

4.2.3 . Encoding Min-cut

To improve efficiency and reduce costs (of installing delay blocks), we calculate a minimum number of transitions, to which the addition of 1-valued delay blocks will make disappear all existing critical pairs. To do this, we encode the problem in SMT by using the max-flow min-cut theorem.

In graph theory, a cut of a given flow network partitions the set of nodes into two disjoint subsets.

Definition 16 *(Min-cut) Given a flow network $FN = (V, E)$ with the source node ι and the target node t, an $\iota - t$ cut (S, T) is a partition of V such that $\iota \in S$ and $t \in T$. The corresponding cutting edges $X_{(S,T)}$ are:*

$$X_{(S,T)} := (S \times T) \cap E = \{(u, v) \in E \mid u \in S, v \in T\}.$$

The capacity $c(S, T)$ of a cut (S, T) is the total weight of the corresponding cutting edges:

$$c(S, T) := \Sigma_{(u,v)\in X_{(S,T)}}\ w(u, v).$$

A min-cut of FN is a cut whose capacity is minimum over all cuts of FN.

For an $\iota - t$ cut (S, T), if we remove all the cutting edges of (S, T), no path exists from ι to t and consequently there is no positive flow from ι to t. Considering the diagnosability problem, the idea is to find a minimum set of transitions in the normal part of all critical pairs such that adding delay blocks can eliminate these critical pairs simultaneously.

Theorem 4 *(max-flow min-cut theorem) In a flow network, the value of a maximum flow passing from the source node to the target node is equal to the capacity of a min-cut.*

We now sketch the basic idea for computing the set of cutting edges of a min-cut, i.e., a min-cut set:
(i) Computing the value of a max-flow based on Ford-Fulkerson algorithm (whose complexity is polynomial [62]), which is equal to the capacity of a min-cut by the max-flow min-cut theorem and equal also to the number of all the cutting edges, as each one has capacity 1.
(ii) Calculating the cutting edges of a min-cut through SMT encoding by using the max-flow computed at the precedent step (this hitting set problem is known to be NP-complete).

Now we show how to build an SMT formula Ψ, whose any satisfying assignment corresponds to a min-cut set. Ψ is composed of different parts to provide a comprehensive description. Given a flow network FN of a given FSA, we first encode the essential static parts and then the S and T sets by integers 0 and 1 through a function $C : V \rightarrow \{0, 1\}$, where integer 0 encodes the nodes in S and 1 the nodes in T.

Initialization

The source state (ι) and the target state (t) should be in S and T, respectively.

$$\Phi^{Init} := ((C(\iota) = 0) \wedge (C(t) = 1))$$

Uniqueness

We formalize here that every state is in S or in T. In addition, due to the way an SMT codes a function, a variable cannot be assigned different values simultaneously, that is, a state cannot both belong to S and T.

$$\Phi^{Uniqueness} := (\bigwedge_{q \in V} (C(q) = 0 \vee C(q) = 1))$$

Weight of the edges

The weights of the edges are valued by 0 or 1 through a function $CE : E \rightarrow \{0, 1\}$. For any edge $(q, q') \in E$, with $q \in S$ and $q' \in T$, this edge is in the cut set

and its weight is 1. Otherwise, its weight is 0.

$$\Phi^{Weight} := (\bigwedge_{(q,q' \in E)} (C(q) = 0 \wedge C(q') = 1 \Rightarrow CE(q,q') = 1) \wedge (C(q) = 1 \vee C(q') = 0 \Rightarrow CE(q,q') = 0))$$

Max-flow

In the calculation process, at each step i, we calculate a new edge, whose weight is added to the current flow. For a min-cut, the latter must be less than or equal to the max-flow, denoted by M, and previously computed. And in this case, based on the max-flow min-cut theorem, the current flow computed at the last step is equal to M.

$$\Phi^{M-flow} := (\bigwedge_{i=0}^{n-1} (process(i) \leqslant M))$$

where n is the number of edges.

Here the function $process(i)$ denotes the current number of edges in the cut in construction at step i, which is updated based on the weight of the current edge $(q, q') \in E$ chosen at step i and the previous number. This function is initialized to 0:

$$\Phi^{Update} := ((\bigwedge_{i=1}^{n-1} (process(i) = (process(i-1) + CE(q,q')))) \wedge (process(0) = 0))$$

SMT formula encoding min-cut

We have formalized all conditions, whose satisfying assignment can be easily proved to correspond to a min-cut set. Thus we have:

$$\Psi := \Phi^{Init} \wedge \Phi^{Uniqueness} \wedge \Phi^{Weight} \wedge \Phi^{M-flow} \wedge \Phi^{Update}$$

and the min-cut set corresponding to any given solution is obtained as the set of the edges (q, q') such that $CE(q, q') = 1$. Note that a min-cut set is not unique, but the formula Ψ ensures that for every path from ι to t, representing a critical pair, at least one of its transitions will be counted in the min-cut set.

Example 8 *For the unfolded T_G^C of the Figure 4.6, there are four min-cut sets of size one, corresponding to its four transitions.*

Applied to the flow network FN which is the unfolding of the critical twin plant T_G^C of the system G, a min-cut set is thus made up of pairs of transitions in G (each one corresponding to observable transitions in the faulty path and normal path, respectively, of some critical pair). The idea is thus to differ one transition of such each pair by adding to it a 1-valued delay block. Now, it may happen

that both transitions of a pair are equal. This is the case of $(q_0, o1, q_1)$ from the example depicted in Figure 2.2, which appears twice in the first transition (issued from the initial state) of the corresponding T_G^C of Figure 4.4 and thus in the first transition of its unfolded FN of Figure 4.6. Obviously, adding a delay block to such a transition would have the same effect on the normal and faulty event sequences of the concerned critical pair, that will thus not be disambiguated. So, all min-cut sets containing such a pair of identical transitions have to be discarded as disambiguating all critical pairs from such a min-cut set is impossible. For the example, it means that the min-cut set corresponding to the first transition of the FN of Figure 4.6 is discarded and thus only three min-cut sets remain. In practice, the transitions of the critical twin plant T_G^C that are made up of a pair of identical transitions are eliminated when constructing its unfolded FN by the analog of the delay closure operation by considering these transitions as silent ones. In the example, this means that the first transition is removed from FN in Figure 4.6 (transition labeled by $o2$ goes directly from state ι to state $((q_2, N), (q_5, N))$) and there are thus only three possible min-cut sets of size one.

Now, it is reasonable, in a pair of transitions of G constituting an element of a min-cut set, to choose to add a 1-valued delay block to the transition belonging to the normal path (as the normal part of G is the one issued from its design). Thus a solution to our problem of making the system diagnosable will be obtained from any given min-cut set X by adding a 1-valued delay block to all the transitions in G appearing in X as the normal components of the pairs of transitions it is made up of. We denote by δ_X the set of these transitions:

$$\delta_X = \{t = (q, \sigma, q') \in \delta \mid \exists t_F \in \delta\ (t_F, t) \in X\} \text{ where } X = \{e \in E \mid C(e) = 1\} \text{ in a solution of } \Psi \text{ for unfolded } T_G^C$$

Note that it may happen that two distinct transitions of the min-cut set X have the same normal component (this means that this normal transition appears in two different critical pairs). Thus $|\delta_X|$ may be smaller than $|X|$. It may also happen that two different min-cut sets X and X' give rise to the same solution in terms of transitions to be delayed: $\delta_X = \delta_{X'}$. Actually, if by construction there is no cycle in FN, which is the unfolded T_G^C, it may occur cycles, and thus several occurrences of a same transition of G, in the normal path corresponding to a path from ι to t in FN, thus a same transition of G may appear several times as a candidate to block some critical pair.

Example 9 *This is the case of the example as the third and fourth transitions (both labeled $o1$) of Figure 4.6 give rise to the same normal transition $(q_5, o1, q_5)$ of G (see Figure 2.2) to delay. The second transition (labeled $o2$) provides $(q_3, o2, q_5)$. There are thus only two ways of adding a 1-valued delay block to a transition of G to make G diagnosable (the second gives the ADB of Figure 4.5).*

4.2.4 . Diagnosability Conditions

We have seen that it was necessary to remove transitions in FN (the unfolded T_G^C) that are made up of a pair of identical transitions. But this elimination may create an empty path in FN between ι and t, meaning that the system cannot be made diagnosable this way. In fact, our method fails if and only if, after elimination of those common transitions, FN contains an empty path. The simplest prototypical case of such an impossibility is given when G is made up of faulty and unobservable transitions, both from the initial state q_0 to a state q_1, followed by an observable loop in q_1. The latter transition is common to both normal and faulty paths of the existing critical pair, which thus cannot be eliminated. Actually, all the cases where the method fails are generalizations of the above-mentioned one.

Note also that a loop of an observable transition, so an infinite succession of a same observable event, in a critical pair does not pose a problem only if we take care to consider timed observations, i.e., to distinguish a same observable event occurring at time 0 (without delay) and at time 1 (with a delay): adding a delay block to such a transition does not change the trajectory observed when considering only order but changes it when also considering time. This is the case in our example (see Figure 2.2) when adding a delay block to $(q_5, o1, q_5)$: the untimed normal trajectory remains identical, and the same as the untimed faulty trajectory, i.e., $o1o2o1^*$, while the timed normal trajectory becomes $o1(0)o2(0)o1(1)^* = o1o2$ $1\ o1^*$, different from the timed faulty trajectory $o1(0)o2(0)o1(0)^* = o1o2o1^*$.

In the same way, in the absence of common transitions in the normal and faulty diagnosers, adding delay blocks cannot create new critical pairs when considering timed observations because all faulty trajectories have their observable events occurring at time 0.

4.3 . Implementation and Validation

To show the feasibility of our approach, we carried out a prototype implementation done in Python by using the SMT solver Z3. All our experimental results are obtained by running our programs on a Mac OS laptop with the processor 2.7 GHz Intel Core i5, 8 Go 1600 MHz DDR3 of memory. Source code and experiments are available at https://github.com/lu-1993/Designing-DIA-via-delay-blocks.

We reported on several versions of five benchmarks from literature, which are all non-diagnosable DESs. Furthermore, considering that such literature examples are normally quite small, to study the scalability, we tested also some hand-crafted versions whose state space was generated in a partially random way while keeping the verdicts. The first one concerns the example shown in Figure 2.2. The second is about the HVAC system from [82]. The third is a modified model from [92]. The fourth is an example from [32]. And the last one was presented in [59]. Furthermore, ex_2, ex_3, $hvac_2$, $hvac_3$, Su_2, Su_3, $Mehdi_2$, $Mehdi_3$ and $Jiang_2$,

$Jiang_3$ are corresponding hand-crafted versions.

	$Size$	$\|clauses\|$	$\|Cut\|$	$SAT?$	$time(N\_C)$	$time(M\_C)$
ex_1	16	89	3	Yes	15.07	14.72
ex_2	213	1800	3	Yes	15.38	14.86
ex_3	500	19855	3	Yes	100.32	24.57
$hvac_1$	14	191	2	Yes	0.72	0.58
$hvac_2$	114	1920	2	Yes	0.82	0.63
$hvac_3$	500	8576	2	Yes	40.33	10.44
Su_1	14	64	1	Yes	0.27	0.18
Su_2	108	643	1	Yes	0.50	0.61
Su_3	500	6753	1	Yes	18.22	9.83
$Mehdi_1$	12	407	2	Yes	0.70	0.18
$Mehdi_2$	116	3248	2	Yes	0.80	0.65
$Mehdi_3$	500	8975	2	Yes	49.33	12.22
$Jiang_1$	10	36	2	No	0.48	0.03
$Jiang_2$	114	450	2	No	0.63	0.42
$Jiang_3$	500	8675	2	No	28.33	9.22

Table 4.1: Experimental results

Table 4.1 shows part of our experimental results, the 2nd column gives the number of transitions of the system, the 3rd shows the size of the formula expressed by its number of clauses when min-cut value is used, the 4th is the min-cut value, obtained by pre-processing, which equals the max-flow value, the 5th is the satisfiability verdict, the last two are the execution times in seconds when min-cut value is used or not used (in this case, a cut is computed without size condition and then a cut of smaller size that the one just obtained is searched and the process is iterated up to having no solution, in which case the last cut obtained is a min-cut), respectively. As can be seen, our approach is feasible in practice as Z3 can get the min-cut set in the time we specified (set to 1500 seconds). In addition, it shows that the use of min-cut value (computed in advance as the value of max-flow) improves the efficiency, which justifies our strategy. It is worth noting that we have carefully chosen and modified the examples, $Jiang_i$, for which our approach cannot work for the reason that we have explained in Section 4.2.4.

4.4 . Related Work

After introducing the diagnosability definition of DESs, a fair amount of research has dealt with how to guarantee this property under limited sensor capacities, which is an important decision-making problem for automated systems. One well-known approach called active diagnosis has been initially proposed by [83].

Precisely, if a given system is not diagnosable, then a partial-observation controller is synthesized in order to force the system to stay within a diagnosable subset of its behaviors. An active diagnoser is composed of the controller and the diagnoser. After that, [33] has proposed another planning-based approach by constructing a twin plant. Then [53] has proved that the active diagnosability problem is EXPTIME-complete and proposed a way to synthesize a memory-optimal active diagnoser.

Another way to handle this problem is to calculate a subset of observable events with minimum cardinality to satisfy diagnosability, provided that the original observable set is enough to guarantee this property. This problem has been proved to be NP-complete by [103]. Then followed some work on designing a mimimal sensor set for diagnosability, that contains the original sensor set when the system is not diagnosable ([28, 16]). However, its cardinality could be quite large and thus not very practical in reality.

Different from the above methods, our approach is applied on the existing non-diagnosable system to make it diagnosable with delay blocks in a non-intrusive way, whose complexity is NP.

4.5 . Conclusion

Given a non-diagnosable system, we have shown how to find a relatively small set of transitions such that adding delay blocks to them can disambiguate all critical pairs without generating new ones. This is done by using max-flow min-cut theorem, encoded in SMT. We have shown the efficiency of our algorithm with benchmarks. However, this cannot guarantee to provide a solution for any system. And, for real applications, assuming that a subset of observable events occur at the same time unit, is not realistic, as in practice, event occurrence is never instantaneous. To handle such various occurrence times, an interesting perspective is to investigate how to extend the current approach to general delay blocks, i.e., delay blocks with various delays. This requires computing the delays in order to avoid as far as possible creating new critical pairs.

5 - Manifestability Property and its Verification

In this chapter, we present our last contribution during my PhD studies. After having analyzed diagnosability both for DESs and RTSs, we realized that developing a diagnosable system in real life is sometimes too expensive because it requires a very high number of sensors. Therefore, in order to achieve a trade-off between the cost at design stage and the possibility to observe a fault manifestation, we propose a new system property called manifestability, which makes sure that a faulty system has at least one future observable behavior after some fault occurrence distinguishable from all normal behaviors, while requiring only a reasonable number of sensors. We define formally this property and study how to verify it, for DESs and RTSs successively. We implement our algorithms for FSA and a subclass of TA (the problem being in general not decidable for TA) and provide experimental results that show the feasibility of our approach from a practical point of view. This chapter has been the subject of publications [101, 38].

5.1 . Motivation/Introduction

Fault diagnosis is a crucial and challenging task in the automatic control of complex systems, whose efficiency depends as we have seen on a system property called diagnosability. Recall that this property describes whether one can distinguish with certainty fault behaviors from normal ones based on sequences of observable events emitted from the system. In a given system, the existence of two infinite behaviors with the same observations, where exactly one contains the considered fault, violates diagnosability. Following our previous work on diagnosability analysis of discrete-event systems based on SAT and real-time systems based on SMT, we are able to design quite secure systems. However, in reality, diagnosability turns out to be a quite strong property that generally requires a high number of sensors. Consequently, it is often too expensive to develop a diagnosable system.

To achieve a trade-off between the cost, i.e., a reasonable number of sensors, and the possibility to observe a fault manifestation, we introduce a new property that we call manifestability, a term borrowed from philosophy: "... which I shall call the 'manifestability of the mental', that if two systems are mentally different, then there must be some physical contexts in which this difference will display itself in differential physical consequences" [71]. In the domain of diagnosis, similarly, the manifestability property describes the capability of a system to manifest a fault occurrence in at least one future behavior. It is the weakest property a system model should have to have a chance to identify a fault occurrence and it should be analyzed at design stage on the system model. The fault will then necessarily

show itself with nonzero probability after enough runs of the system under the assumption that no behavior described in the model has zero probability. For diagnosability, each future behavior of all fault occurrences should be distinguishable from all normal behaviors, which is a strong property and thus requires high sensor demanding, and this is where diagnosability differs from manifestability. Obviously one has to continue to rely on diagnosability for online safety requirements, i.e., for those faults which may have dramatic consequences if they are not surely detected shortly after they occur, in order to trigger corrective actions. But for all other faults that do not need to be detected at their first occurrence (e.g., whose consequence is a degraded but acceptable functioning that will require maintenance actions in some near future), manifestability checking, which is cheaper in terms of sensors needed, is enough under the fairness assumption above.

5.1.1 . Motivating Example

Now, we explain why it is worth analyzing the manifestability property with a motivating example.

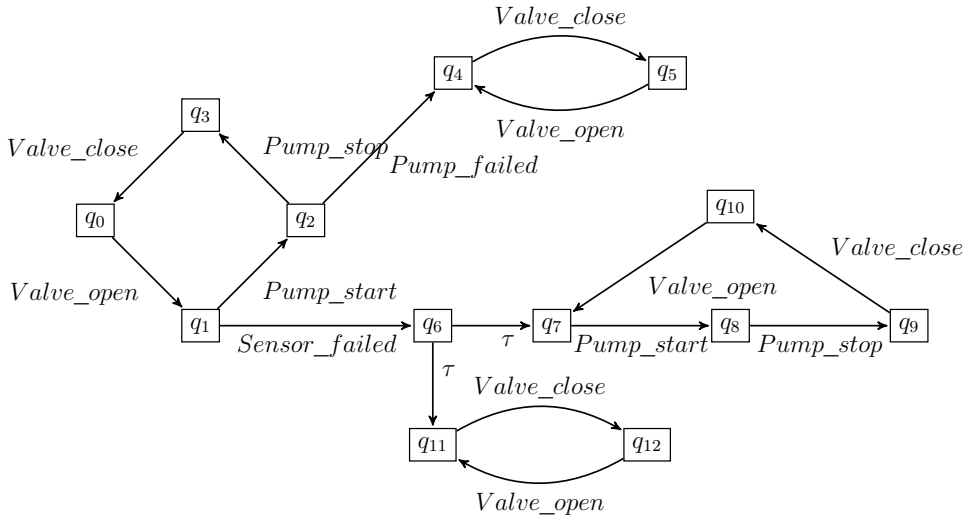

Figure 5.1: A simplified HVAC system.

Example 10 *Figure 5.1 shows a modified version of the HVAC system from [82], which is a composite model that captures the interactions between the three components, i.e., a pump, a valve, and a controller. In this system, the initial state is q_0, the events $Valve\_open$, $Pump\_start$, $Pump\_stop$, $Valve\_close$ are observable and the fault events $Pump\_failed$, $Sensor\_failed$ are not observable, as well as the silent (unobservable normal) event τ, the latter is used to represent non-deterministic behaviors after the occurrence of the fault event $Sensor\_failed$.*

In this system, the infinite correct behavior is $(Valve\_open\ Pump\_start\ Pump\_stop\ Valve\_close)^{\omega}$, *where* ω *denotes the infinite concatenation. After the unobservable fault* $Pump\_failed$, *the system exhibits different observations from the correct behavior, this event is thus diagnosable (see Definition 5). Now consider the other fault event* $Sensor\_failed$, *whose occurrence leads to non-deterministic behaviors: one with the same observations as the correct behavior, the other with different observations. Actually temperature sensor fault can cause valve improperly controlled [15]. Here we consider two independent typical behaviors: either entering* q_7 *by only degrading the efficiency of energy consumption while the system can still assume its basic functionality, or entering* q_{11}, *where the valve closes immediately without executing the pump, leading to observations that are distinguishable from the correct behavior. Suppose that the issue of energy consumption is not the priority of diagnosis. Hence the fact that the fault* $Sensor\_failed$ *can be detected only when it makes enter the state* q_{11} *is still acceptable in this case. The original (stochastic) diagnosability property is not suitable to handle such situations. If we consider manifestability, the fault* $Sensor\_failed$ *is effectively manifestable since its occurrence has at least one future that is distinguishable from the correct behavior.*

5.2 . Manifestability Analysis for Discrete Event Systems

In this section, we first define the manifestability for finite state automata and provide a sufficient and necessary condition with an algorithm based on equivalence checking of languages. Then, we show that the manifestability problem itself is PSPACE-complete; finally we give experimental results to show the practical efficiency of the algorithm.

5.2.1 . Manifestability Property for DESs

We model a DES as a finite state automaton defined in Chapter 2 (see Definition 2). Similar to diagnosability, the manifestability algorithm we will propose would have exponential complexity in the number of fault types (i.e., fault labels, or more generally a partition of fault labels) if all those fault types were considered at once. As in [73, 85], to reduce it to linear complexity in the number of fault types, we consider only one fault type at a time. However, multiple occurrences of faults of the given type are allowed. The faults of other types are processed as unobservable normal events. This is justified as the system is manifestable iff it is manifestable for each fault type. In the following, $\Sigma_f = \{F\}$, where F is the currently considered fault type.

Example 11 *The top part of Figure 5.2 shows an example of a system model* G, *where* $\Sigma_o = \{o1, o2, o3\}, \Sigma_u = \{u1, u2\}$, *and* $\Sigma_f = \{F\}$. *Notice that for diagnosis*

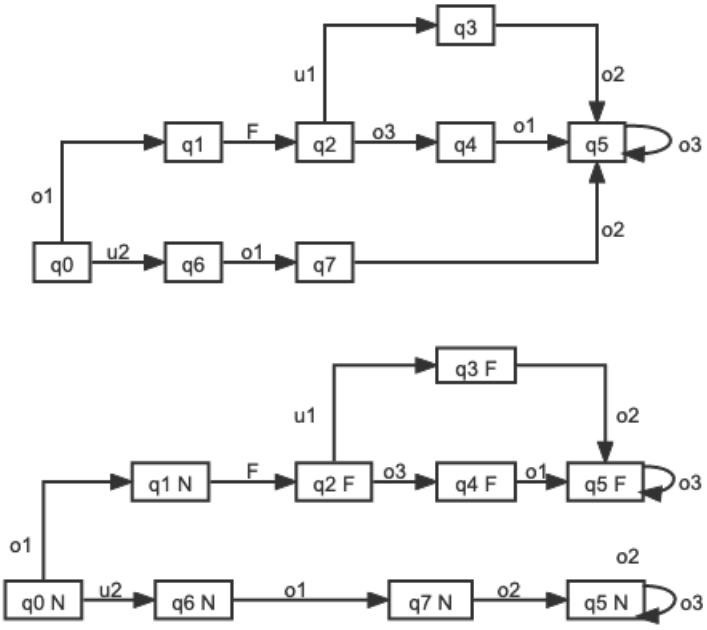

Figure 5.2: A system model (top) and its diagnoser (bottom)

problem, fault is predefined as an unobservable event in the model. This is different from testing, where faulty behaviors are judged against a specification.

Given a system model G, and $L(G)$ its language , i.e., the set of words produced by G, we denote by $L_F(G)$ (resp., $L_N(G)$) those words containing (resp., not containing) F. Traditionally, we do the Assumptions 1 and 2 (see subsection 2.3.1) about the possibility to always continue a trajectory and to observe infinitely the infinite trajectories.

We will need some infinite objects. We denote by Σ^ω the set of infinite words on Σ. We define in an obvious way infinite paths in G and thus $L^\omega(G)$ the language of infinite words recognized by G in the sense of Büchi automata. As all states of G are considered as final states, those infinite trajectories are just the labels of infinite paths, and the concept of Büchi automaton coincides with that of Muller automaton, which can be determinized, according to the McNaughton theorem. We can conclude from this that $L^\omega(G)$ is the set of infinite words whose prefixes belong to $L(G)$ and that two equivalent system models G_1 and G_2, i.e., such that $L(G_1) = L(G_2)$, define the same infinite trajectories, i.e., $L^\omega(G_1) = L^\omega(G_2)$. Particularly, we use $L_F^\omega(G) = L^\omega(G) \cap \Sigma^* F \Sigma^\omega$ for the set of infinite faulty trajectories, and $L_N^\omega(G) = L^\omega(G) \cap (\Sigma \backslash \{F\})^\omega$ for the set of infinite normal trajectories, where $\backslash$ denotes set subtraction. In the following, we use the classical synchronization operation on $\Sigma_s \subseteq \Sigma$ between two automata G_1 and G_2 with events Σ, denoted by $G_1 \| _{\Sigma_s} G_2$, i.e., any event in Σ_s should be synchronized

while others can occur whenever possible.

The following basic operation is aimed at keeping only information about a given set of events. It boils down to replace by ϵ the events not concerned and eliminate the ϵ-transitions (silent transitions) thus created. It will be used to simplify some intermediate structures when checking manifestability without affecting the validity of the result obtained.

Definition 17 (Delay Closure) *Given an automaton $G = (Q, \Sigma, \delta, q_0)$, its delay closure with respect to Σ_d, with $\Sigma_d \subseteq \Sigma$, is $\mathbb{C}_{\Sigma_d}(G) = (Q_d, \Sigma_d, \delta_d, q_0)$, where:*

1. $Q_d = \{q_0\} \cup \{q \in Q \mid \exists s \in \Sigma^*, \exists \sigma \in \Sigma_d, (q_0, s\sigma, q) \in \delta\}$;
2. $(q, \sigma, q') \in \delta_d$ *if* $\sigma \in \Sigma_d$ *and* $\exists s \in (\Sigma \backslash \Sigma_d)^*, (q, s\sigma, q') \in \delta$.

We saw that, for designing a diagnosable (Definition 5) system, each faulty trajectory should be distinguished from all normal trajectories, which is often very expensive in terms of number of sensors required. To reduce such a cost and still make it possible to show the fault after enough runs of the system, we propose now another system property that we call manifestability, which is much weaker than diagnosability. Intuitively, manifestability describes whether or not a fault occurrence has the possibility to manifest itself through observations. More precisely, if a fault is not manifestable, then we can never be sure about its occurrence no matter which trajectory is executed after it. Thus, the system model should be necessarily revised.

Definition 18 (Manifestability of FSA) *F is manifestable in a system model as a finite state automaton G iff*

$$\exists s \in L_F(G), \forall p \in L(G), P(p) = P(s) \Rightarrow F \in p.$$

F is manifestable iff there exists at least one faulty trajectory s in G such that every trajectory p that is observably equivalent to s should contain F. In other words, manifestability is violated iff each occurrence of the fault can never manifest itself in any future. This can be rephrased in terms of diagnosis. Let $Diag$ be the diagnosis procedure with input an observation in Σ_o^* and output a diagnosis in $\{N, F, \{N, F\}\}$. Then, F is manifestable in G iff there exists a trajectory s in G such that $\mathrm{Diag}(P(s)) = \{F\}$, i.e., the correct diagnosis of the occurrence of F can be made for at least one faulty trajectory. This emphasizes that manifestability is actually the weakest requirement for the existence of a useful (i.e., not always ambiguous from any observed faulty trajectory) diagnosis procedure.

Theorem 5 *A fault F is manifestable in a system model G iff one or the other of the following equivalent conditions is satisfied (where $\nprec_{|t|}$ and $\nprec$ denote critical pairs, see Definition 6):*

$$(\mathcal{M}) \quad \exists s^F t \in L_F(G),\ \nexists s' \in L_N(G), s^F t \nprec_{|t|} s',$$

$$(\mathcal{M}_\omega) \quad \exists s \in L_F^\omega(G), \nexists s' \in L_N^\omega(G), s \nsucc s'.$$

Proof 1 *Condition $\mathcal{M}$ is a straightforward rephrasing of Definition 18. We demonstrate condition $\mathcal{M}_\omega$.*
$\Rightarrow$ Suppose that F is manifestable in G. Thus from Definition 18, $\exists s \in L_F(G)$ such that $\nexists s' \in L_N(G)$ with $P(s) = P(s')$. By extending s with enough events, which is possible since the system is living, we obtain then $\exists s \in L_F^\omega(G)$, $\nexists s' \in L_N^\omega(G)$, such that $s \nsucc s'$.
$\Leftarrow$ Suppose now that F is not manifestable in G and show that the condition $\mathcal{M}_\omega$ is consequently not true. From non-manifestability of F and Definition 18, we have $\forall s \in L_F(G)$, $\exists p \in L_N(G), P(p) = P(s)$. This can be formulated as equality of the languages of two automata, as it will be seen in section 5.2.2 ($L_a(V_G) = L_F(D_G^{FR})$). It results that this equality of the languages still holds for infinite words ($L_a^\omega(V_G) = L_F^\omega(D_G^{FR})$), i.e., $\forall s \in L_F^\omega(G)$, $\exists p \in L_N^\omega(G)$ such that $s \nsucc p$, which is $\neg\mathcal{M}_\omega$, i.e., the condition $\mathcal{M}_\omega$ is not true.

Manifestability expresses whether it is possible to have at least one faulty path in a system that is observably distinct from all normal paths, i.e., that there exists a fault occurrence showing itself in at least one of its futures. Therefore, we can define a strong version of manifestability, which requires that any occurrence of the fault should show itself in at least one of its futures in a similar way. We give now the definition of this strong version of manifestability.

Definition 19 (Strong Manifestability for FSA) *Given a system model G and a fault F:*

1. *given $k \in \mathbb{N}, F$ is strongly k-manifestable in G if*

$$\forall s^F \in L(G), \exists t \in L(G)/s^F, |t| \leqslant k,$$
$$\forall p \in L(G), P(p) = P\left(s^F t\right) \Rightarrow F \in p.$$

2. *F is strongly manifestable in G if*

$$\exists k \in \mathbb{N} \text{ such that } F \text{ is strongly } k\text{-manifestable in } G.$$

F is strongly manifestable if, for each s^F in G (and not just for only one as in Definition 18), there exists at least one extension t of s^F in G, such that every trajectory p in G that is observably equivalent to $s^F t$ should contain F. In other words, each occurrence of F should show itself in at least one of its futures. In terms of the diagnosis procedure Diag, it means that any occurrence s^F of F in G possesses a future t such that $\mathrm{Diag}\left(P\left(s^F t\right)\right) = \{F\}$, i.e., the correct diagnosis of any occurrence of F can be made for at least one future trajectory. In a similar way as Theorem 5, we can prove the following theorem, which provides a sufficient and necessary condition for strong manifestability.

Theorem 6 *Given a system model G and a fault F:*

1. *given $k \in \mathbb{N}$, F is strongly k-manifestable in G iff the following condition is satisfied:*
$$(\mathcal{M}_k^s) \quad \begin{array}{l} \forall s^F \in L(G), \exists t \in L(G)/s^F, |t| \leqslant k, \\ \nexists s' \in L_N(G), s^F t \not\prec_{|t|} s'. \end{array}$$

2. *F is strongly manifestable in G iff one or the other of the following equivalent conditions is satisfied:*
$$\begin{array}{ll} (\mathcal{M}^s) & \forall s^F \in L(G), \exists t \in L(G)/s^F, \\ & \nexists s' \in L_N(G), s^F t \not\prec s', \\ (\mathcal{M}_\omega^s) & \forall s^F \in L(G), \exists t \in L^\omega(G)/s^F, \\ & \nexists s' \in L_N^\omega(G), s^F t \not\prec s'. \end{array}$$

Proof 2 *1. is just a rephrasing of Definition 19.1 and condition $\mathcal{M}^s$ of 2. a rephrasing of Definition 19.2. It is straightforward, by using liveness of $L(G)$, that strong manifestability implies $\mathcal{M}_\omega^s$. Consider the reverse. If F is not strongly manifestable, then $\exists s^F \in L(G), \forall t \in L(G)/s^F, \exists s' \in L_N(G), s^F t \not\prec_{|t|} s'$. This means that any faulty trajectory of prefix s^F in G is equal to a trajectory in the synchronized product of G by itself on observable events (after having erased the unobservable events of the second copy), which can be expressed as languages equality of two automata (see section 5.2.2), which still holds for infinite words, giving $\neg\mathcal{M}_\omega^s$.*

The following theorem provides the relationships between diagnosability, strong manifestability and manifestability.

Theorem 7 *Given a system model G and a fault F, we have:*

1. *F is k-diagnosable (resp., diagnosable) in G implying that F is strongly k-manifestable (resp., strongly manifestable) in G.*

2. *F is strongly manifestable in G implying that F is manifestable in G.*

Proof 3 *1. Suppose that F is not strongly manifestable, then from Theorem 6, we have $\neg\mathcal{M}_\omega^s$, i.e., $\exists s^F \in L(G), \forall t \in L^\omega(G)/s^F$, $\exists s' \in L_N^\omega(G)$ such that $s^F t \not\prec s'$. This implies that there does exist at least one critical pair in the system. From Theorem 1 , F is not diagnosable (the proof is similar for k).*

2. *Suppose that F is not manifestable. From Theorem 5, we have $\forall s \in L_F^\omega(G)$, $\exists s' \in L_N^\omega(G)$, such that $s \not\prec s'$. By choosing arbitrarily one $s^F \in L(G)$ and taking all s of prefix s^F, we obtain $\exists s^F \in L(G), \forall t \in L^\omega(G)/s^F$, $\exists s' \in L_N^\omega(G)$ such that $s^F t \not\prec s'$, i.e., $\neg\mathcal{M}_\omega^s$. Hence F is not strongly manifestable.*

5.2.2 . Manifestability Verification

Manifestability verification consists in checking whether the condition $\mathcal{M}_\omega$ (or $\mathcal{M}$) in Theorem 5 is satisfied for a given system model. In this section, we show how to construct different structures based on a system model to obtain $L_F^\omega(G)$, $L_N^\omega(G)$ as well as the set of critical pairs. The condition $\mathcal{M}_\omega$ (or $\mathcal{M}$) can then be checked by using equivalence techniques with these intermediate structures.

System Diagnosers

Given a system model, the first step is to construct a structure showing fault information for each state, i.e., whether the fault has effectively occurred up to this state from the initial state.

Definition 20 (Diagnoser) *Given a system model G, its diagnoser with respect to a considered fault F is the automaton $D_G = (Q_D, \Sigma, \delta_D, q_0)$, where:*

- $Q_D \subseteq Q \times \{N, F\}$ *is the set of states;*
- Σ *is the set of events;*
- $\delta_D \subseteq Q_D \times \Sigma_D \times Q_D$ *is the set of transitions;*
- $q_0 = (q_0, N)$ *is the initial state.*

The transitions of δ_D are those $((q, \ell), e, (q', \ell'))$, with (q, ℓ) reachable from q_0 such that there is a transition $(q, e, q') \in \delta$, and $\ell' = F$ if $\ell = F \vee e = F$, otherwise $\ell' = N$.

Example 12 *The bottom part of Figure 5.2 shows the diagnoser for the system depicted in the top part, where each state has its own fault information. More precisely, given a system state q, if the fault has occurred in all paths from q_0 to q, then the fault label for q is F. Such a state is called fault (diagnoser) state. If the fault has not occurred in any path from q_0 to q, then the fault label for q is N and the state is called normal (diagnoser) state. Otherwise q is split into two states, one labeled with F and the other with N (q_5 in the example). Diagnoser construction keeps the same set of trajectories and splits into two (fault state and normal state) those states reachable by both a faulty and a normal path (q_5 in the example).*

Lemma 1 *Given a system model G and its corresponding diagnoser D_G, then we have $L(G) = L(D_G)$ and $L^\omega(G) = L^\omega(D_G)$.*

In order to simplify the automata handled, the idea is to keep only the minimal subparts of D_G useful for the following manifestability checking, while containing all faulty (resp., normal) trajectories.

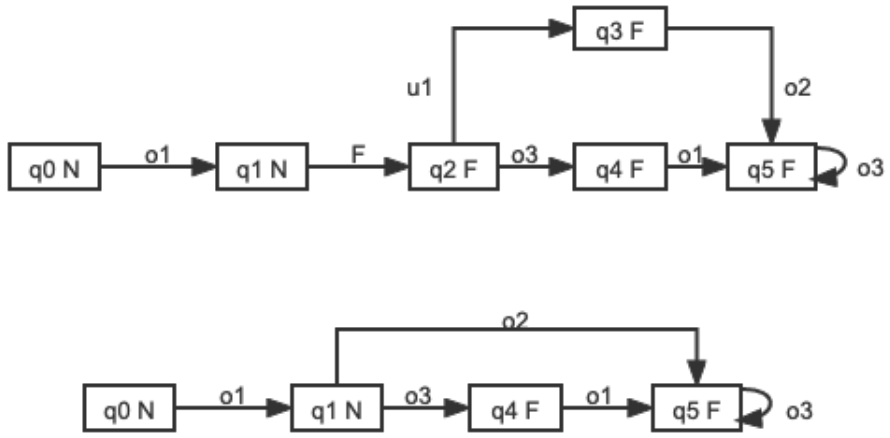

Figure 5.3: Fault diagnoser (top) and its refined version (bottom) for Example 11

Definition 21 (Fault (Refined) Diagnoser) *Given a diagnoser D_G, its fault diagnoser is the automaton $D_G^F = (Q_{D^F}, \Sigma_{D^F}, \delta_{D^F}, q_0)$, where:*

1. $Q_{D^F} = \{q_D \in Q_D \mid \exists q'_D = (q, F) \in Q_D, \exists s' \in \Sigma^*, (q_D, s', q'_D) \in \delta_D^*\}$;
2. $\delta_{D^F} = \left\{\left(q_D^1, \sigma, q_D^2\right) \in \delta_D \mid q_D^2 \in Q_{D^F}\right\}$;
3. $\Sigma_{D^F} = \{\sigma \in \Sigma \mid \exists (q_D^1, \sigma, q_D^2) \in \delta_{D^F}\}$.

The fault refined diagnoser is obtained by performing the delay closure with respect to the set of observable events Σ_o on the fault diagnoser: $D_G^{FR} = \mathbb{C}_{\Sigma_o}(D_G^F)$.

The fault diagnoser keeps all reachable fault states as well as all transitions and intermediate normal states on paths from the initial state q_0 to any faulty one. Then we refine this fault diagnoser by only keeping the observable information, which is sufficient to obtain the set of critical pairs. The top (resp., bottom) part of Figure 5.3 shows the fault diagnoser (resp., fault refined diagnoser) for Example 12.

By construction, the sets of faulty trajectories in D_G^F and in G are equal and this is still true for infinite faulty trajectories. This is also the case for faulty trajectories in D_G^{FR} (defined as labels of paths in D_G^{FR} containing a fault state or whose last state reached owns a transition to a fault state and denoted as $L_F(D_G^{FR})$) and observed faulty trajectories in G (finite or infinite). But take care that it may exist infinite normal trajectories in D_G^F (resp., D_G^{FR}) if it exists in G a normal cycle in a path to a first fault state (e.g., adding a loop in state q_1 of the system model of Example 12).

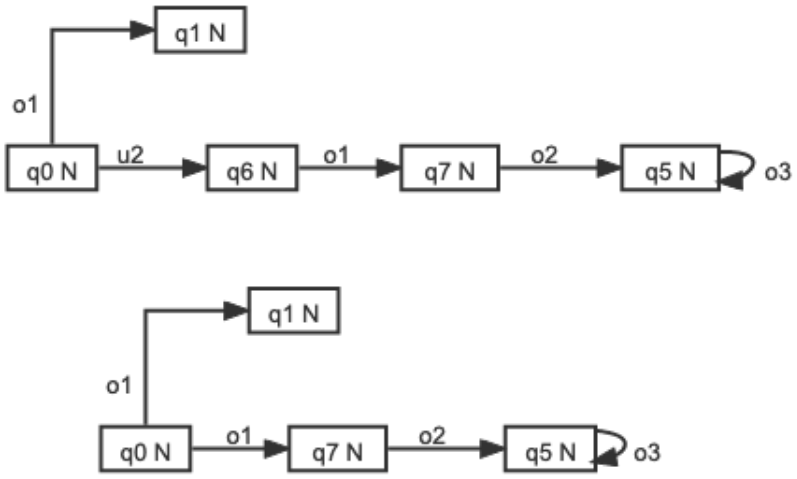

Figure 5.4: Normal diagnoser (top) and its refined version (bottom) for Example 11

Lemma 2 *Given a system model G and its corresponding fault diagnoser D_G^F and fault refined diagnoser D_G^{FR}, we have $L_F(G) = L_F\left(D_G^F\right), L_F^\omega(G) = L_F^\omega\left(D_G^F\right)$ and $P\left(L_F(G)\right) = L_F\left(D_G^{FR}\right), P\left(L_F^\omega(G)\right) = L_F^\omega\left(D_G^{FR}\right)$.*

Similarly, we obtain the subpart of D_G containing only normal trajectories.

Definition 22 (Normal (Refined) Diagnoser) *Given a diagnoser D_G, its normal diagnoser is the automaton $D_G^N = (Q_{D^N}, \Sigma_{D^N}, \delta_{D^N}, q_0)$, where:*

1. $Q_{D^N} = \{(q, N) \in Q_D\}$;
2. $\delta_{D^N} = \left\{\left(q_D^1, \sigma, q_D^2\right) \in \delta_D \mid q_D^2 \in Q_{D^N}\right\}$;
3. $\Sigma_{D^N} = \left\{\sigma \in \Sigma \mid \exists \left(q_D^1, \sigma, q_D^2\right) \in \delta_{D^N}\right\}$.

The normal refined diagnoser is obtained by performing the delay closure with respect to Σ_o on the normal diagnoser: $D_G^{NR} = \mathcal{C}_{\Sigma_o}(D_G^N)$.

The top (resp., bottom) part of Figure 5.4 shows the normal diagnoser (resp., normal refined diagnoser) for Example 12. Note the presence of the deadlock state (q_1, N), showing that $L_N(G)$ is not necessarily alive.

Lemma 3 *Given a system model G and its corresponding normal diagnoser D_G^N and normal refined diagnoser D_G^{NR}, we have $L_N(G) = L\left(D_G^N\right), L_N^\omega(G) = L^\omega\left(D_G^N\right)$ and $P\left(L_N(G)\right) = L\left(D_G^{NR}\right), P\left(L_N^\omega(G)\right) = L^\omega\left(D_G^{NR}\right)$.*

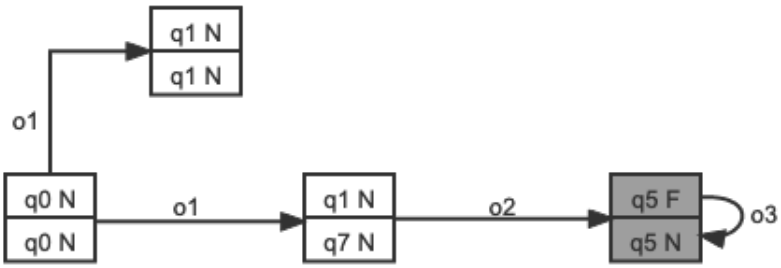

Figure 5.5: The pair verifier for the system in Example 11

Manifestability Checking

In the precedent subsection we have given the definition of different diagnosers. Now we show how to obtain the set of critical pairs based on them. On this basis, an equivalence checking will be used to test the manifestability condition $\mathcal{M}_\omega$ (or $\mathcal{M}$) in Theorem 5.

Definition 23 (Pair Verifier) *Given a system model G, its pair verifier V_G is obtained by synchronizing the corresponding fault and normal refined diagnosers D_G^{FR} and D_G^{NR} based on the set of observable events, i.e., $V_G = D_G^{FR} \|_{\Sigma_o} D_G^{NR}$.*

To construct a pair verifier, all observable events are synchornized. Then V_G is actually the product of D_G^{FR} and D_G^{NR} and the language of the pair verifier is thus the intersection of the language of the fault refined diagnoser and that of the normal refined diagnoser. In the pair verifier, each state is composed of two diagnoser states, among which one is from the fault refined diagnoser and indicates the effective occurrence of the fault in the corresponding trajectory with the label F, otherwise with N. And the other one is from the normal refined diagnoser, so all labels are N. If one of the two labels in a verifier state is F, then this verifier state is called an ambiguous state. The reason is that to reach this state, both trajectories have the same observations with exactly one trajectory containing the fault. Trajectories of V_G are thus either normal (both labels are N) or ambiguous (all state labels from a certain state are (F, N) or the last state reached owns a transition to an ambiguous state), the latter ones being denoted by $L_a(V_G)$ (resp., $L_a^\omega(V_G)$ for infinite ones).

Lemma 4 *Given a system model G with its D_G^{FR}, D_G^{NR} and V_G, we have $L_a(V_G) = L_F\left(D_G^{FR}\right) \cap L\left(D_G^{NR}\right)$ and $L_a^\omega(V_G) = L_F^\omega\left(D_G^{FR}\right) \cap L^\omega\left(D_G^{NR}\right)$.*

In the pair verifier depicted in Figure 5.5 , the gray node represents an ambiguous state.

From Lemmas 2, 3, 4, Definition 6 plus Theorems 1 and 5, it is straightforward to get the two following theorems.

Theorem 8 *Given a system model G, a fault F is diagnosable in G iff $L_a^\omega(V_G) = \varnothing$.*

Proof 4 $L_a^\omega(V_G) \neq \varnothing \Leftrightarrow L_F^\omega(D_G^{FR}) \cap L^\omega(D_G^{NR}) \neq \varnothing$ *(from Lemma 4)* $\Leftrightarrow P(L_F^\omega(G)) \cap P(L_N^\omega(G)) \neq \varnothing$ *(from Lemmas 2 and 3)* $\Leftrightarrow \exists s \in L_F^\omega(G), \exists s' \in L_N^\omega(G)\ P(s) = P(s') \Leftrightarrow \exists s, s' \in L^\omega(G)\ s \not\asymp s'$ *(from Definition 6)* $\Leftrightarrow F$ *is not diagnosable (from Theorem 1).*

Theorem 9 *Given a system model G, a fault F is manifestable in G iff $L_a(V_G) \subset L_F\left(D_G^{FR}\right)$ or, equivalently, iff $L_a^\omega(V_G) \subset L_F^\omega\left(D_G^{FR}\right)$, where $\subset$ is the strict inclusion.*

Proof 5 $L_a^\omega(V_G) \not\subset L_F^\omega(D_G^{FR}) \Leftrightarrow L_F^\omega(D_G^{FR}) \subseteq L^\omega(D_G^{NR})$ *(from Lemma 4)* $\Leftrightarrow P(L_F^\omega(G)) \subseteq P(L_N^\omega(G))$ *(from Lemmas 2 and 3)* $\Leftrightarrow \forall s \in L_F^\omega(G), \exists s' \in L_N^\omega(G)\ P(s) = P(s') \Leftrightarrow \forall s \in L_F^\omega(G), \exists s' \in L_N^\omega(G)\ s \not\asymp s'$ *(from Definition 6)* $\Leftrightarrow \neg\mathcal{M}_\omega \Leftrightarrow F$ *is not manifestable (from Theorem 5). The proof is identical when using finite trajectories and property $\mathcal{M}$.*

From this Theorem, it follows that the system in Example 11 is manifestable as Figure 5.3 and Figure 5.5 show that $L_F^\omega(D_G^{FR}) \backslash L_a^\omega(V_G) = \{o_1 o_3 o_1 o_3^\omega\}$.

Adapting the proof of Theorem 9 by using Theorem 6 instead of Theorem 5, i.e., by reasoning on property $\mathcal{M}_\omega^s$ (or equivalently $\mathcal{M}^s$) instead of property $\mathcal{M}_\omega$ (or equivalently $\mathcal{M}$), one obtains:

Theorem 10 *Given a system model G, a fault F is strongly manifestable in G iff $\forall s^F \in L(G), L_a(V_G) \cap P(s^F)\Sigma_o^* \subset L_F(D_G^{FR}) \cap P(s^F)\Sigma_o^*$ or, equivalently, $L_a^\omega(V_G) \cap P(s^F)\Sigma_o^\omega \subset L_F^\omega(D_G^{FR}) \cap P(s^F)\Sigma_o^\omega$.*

So, when manifestabilty requires only strict inclusion of the language $L_a(V_G)$ into the language $L_F(D_G^{FR})$, strong manifestability requires that this strict inclusion holds for all corresponding sub-languages made up of the words of both languages having a given prefix equal to the observation of an arbitrary trajectory ending by an occurrence of F. Conversely, to verify non-strong manifestability, it is enough to find one fault trajectory s^F such that there is equality of both sub-languages made up of words of prefix $P(s^F)$: $L_a(V_G) \cap P(s^F)\Sigma_o^* = L_F(D_G^{FR}) \cap P(s^F)\Sigma_o^*$.

Algorithm

Now, we give the pseudo-code as Algorithm 1 to verify manifestability, which can verify diagnosability simultaneously.

Given the input as the system model G and the fault F, we first construct the diagnoser (line 1) as defined by Definition 20. We then construct the fault and normal refined diagnosers (lines 2-3) as defined by Definitions 21 and 22. Next, we obtain the pair verifier V_G (line 4) by synchronizing D_G^{FR} and D_G^{NR}. With D_G^{FR} and V_G, we have the following verdicts:

Algorithme 1 : Manifestability and Diagnosability Algorithm for DESs

Input : System model G; the considered fault F

1 $D_G \leftarrow$ ConstructDiagnoser (G);
2 $D_G^{FR} \leftarrow$ ConstructFRDiagnoser (D_G);
3 $D_G^{NR} \leftarrow$ ConstructNRDiagnoser (D_G);
4 $V_G \leftarrow D_G^{FR} \|_{\Sigma_o} D_G^{NR}$;
5 **if** $L_a^\omega(V_G) = \varnothing$ **then**
6 | return "F is diagnosable and manifestable in G"
7 **else if** $L_a^\omega(V_G) = L_F^\omega\left(D_G^{FR}\right)$ *(or, equivalently,* $L_a(V_G) = L_F(D_G^{FR})$*)* **then**
8 | return "F is neither diagnosable nor manifestable in G"
9 | **else**
10 | return "F is not diagnosable but manifestable in G";

- if $L_a^\omega(V_G) = \varnothing$ (line 5), F is diagnosable and thus manifestable (line 6).
- if $L_a^\omega(V_G) = L_F^\omega\left(D_G^{FR}\right)$ or, equivalently, $L_a(V_G) = L_F(D_G^{FR})$ (line 7), both nonempty necessarily, F is not manifestable and thus not diagnosable (line 8).
- if $L_a^\omega(V_G) \neq \varnothing$ and if $L_a^\omega(V_G) \subset L_F^\omega\left(D_G^{FR}\right)$ or, equivalently, $L_a(V_G) \subset L_F\left(D_G^{FR}\right)$ (line 9), F is not diagnosable but manifestable (line 10).

Note that, as the infinite normal trajectories are identical in V_G and in D_G^{FR} (and idem for finite trajectories), the condition $L_a^\omega(V_G) = L_F^\omega\left(D_G^{FR}\right)$ is equivalent to $L^\omega(V_G) = L^\omega\left(D_G^{FR}\right)$. Note also that $L_F^\omega\left(D_G^{FR}\right) = L^\omega\left(D_G'^{FR}\right)$ (resp., $L_a^\omega(V_G) = L^\omega(V_G')$) where $D_G'^{FR}$ is identical to D_G^{FR} (resp., V_G' identical to V_G), except that the final states, for Büchi acceptance conditions, are limited to fault (resp., ambiguous) states.

Complexity

In algorithm 1, the complexity of any diagnoser construction is polynomial. When we synchronise the fault and the normal refined diagnosers to obtain the pair verifier, the complexity is polynomial in the number of system states. Finally, when we check the manifestability condition, the language equivalence checking (line 7) is known to be a PSPACE problem (even for infinite words). Thus, the total complexity of algorithm 1 is PSPACE. This algorithm suggests that the manifestability problem is more complex than diagnosability, for which a test of language emptiness is sufficient (line 5), which implies a total NLOGSPACE complexity (in fact it is a result already known that checking diagnosability is NLOGSPACE-complete). In fact, we have shown [38] that the manifestability verification problem itself is

PSPACE-complete by the reduction to it of rational language equivalence checking, known to be PSPACE-complete.

Theorem 11 *Given a system model G and a fault F, the problem of checking whether F is manifestable in G is PSPACE-complete.*

Proof 6 *The complexity of Algorithm 1 is PSPACE. We show that the problem of checking manifestability is PSPACE-hard. Let $G_1 = (Q_1, \Sigma, \delta_1, q_0^1)$ and $G_2 = (Q_2, \Sigma, \delta_2, q_0^2)$ be two arbitrary (non-deterministic) automata on the same vocabulary defining prefix-closed live languages. One can always assume that $Q_1 \cap Q_2 = \emptyset$. Based on G_1 and G_2, one can construct a new automaton, representing a system model, $G = (Q, \Sigma \cup \{\tau, F\}, \delta, q_0)$, where $Q = Q_1 \cup Q_2 \cup \{q_0\}$ and $\delta = \delta_1 \cup \delta_2 \cup \{(q_0, F, q_0^1), (q_0, \tau, q_0^2)\}$, with $\Sigma_o = \Sigma$, $\Sigma_u = \{\tau\}$ and $\Sigma_f = \{F\}$. From the construction of G, one has $L(G_1) = P(L_F(G))$ and $L(G_2) = P(L_N(G))$. From Lemmas 2, 3 and 4, one obtains $L_a(V_G) = P(L_F(G)) \cap P(L_N(G))$. This implies $L(G_1) \cap L(G_2) = L_a(V_G)$. From Theorem 9, one has $L(G_1) \cap L(G_2) \subset L(G_1) \iff F$ is manifestable in G, i.e., $L(G_1) \subseteq L(G_2) \iff F$ is not manifestable in G. So, rational languages inclusion testing boils down to manifestability checking, which gives the result. Note that we could do exactly the same proof using the languages of infinite words and again Theorem 9, and the fact that the problem of checking non-deterministic automata equivalence on infinite words has also been proved to be PSPACE-complete [89]. And note that the proof shows also that checking strong manifestability is PSPACE-hard.*

Let now consider verifying strong manifestability. It is obvious from Definition 19 that F is strongly manifestable in G iff each occurrence of F as a transition label is strongly manifestable in G. We can thus assume that there is only one transition in G labeled by F, say (q_F, F, q'_F). From Theorem 10, proving non-strong manifestability of F in G is equivalent to find $s^F \in L(G)$ such that $L_F(D_G^{FR}) \cap P(s^F)\Sigma_o^* \subseteq L_a(V_G) \cap P(s^F)\Sigma_o^*$. In order to simplify the notations, assume in this paragraph that the fault refined diagnoser D_G^{FR} is obtained by the delay closure with respect to $\Sigma_o \cup \{F\}$, i.e., decide to keep the event F in D_G^{FR} and thus in V_G too (this changes nothing to statement of Theorem 10). So, we check the existence of $sF \in L_F(D_G^{FR})$ such that:

$$L_F(D_G^{FR}) \cap sF\Sigma_o^* \subseteq L_a(V_G) \cap sF\Sigma_o^*. \quad \text{(NSM)}$$

Let $\{(q_F, q_i)\}_{i \in I}$ be the set of all states of V_G (trimmed w.r.t. ambiguous states co-accessibility) whose first component is q_F (we omit to write associated fault labels, N – and possibly also F if the F transition is part of a cycle – for q_F and N for the q_i's). Note that q_F appears once among the q_i's. If the property (NSM) is satisfied for some sF, then any extension of sF in $L_F(D_G^{FR})$ has to appear as an extension of sF in $L_a(V_G)$, i.e., $L_{q'_F}(D_G^{FR}) \subseteq \bigcup_{i \in I} L_{(q'_F, q_i)}(V_G)$, where $L_q(G)$ denotes the set of words produced by G from state q, i.e., as if q was the initial state

of G. But this is only a necessary condition, not sufficient in general. Actually, if corresponding extensions in V_G need several (q'_F, q_i) as starting states, (NSM) property to be satisfied requires that a common prefix s exists for all of them, i.e., a common word in the associated languages $L(V_G, (q_F, q_i))$, where $L(G, q)$ denotes the set of words produced from paths from initial state to q in G, i.e., as if q was the only final state (s will then necessarily be a prefix in $L(D_G^{FR}, q_F)$ too). Finally, the existence of sF verifying (NSM) is equivalent to the existence of $J \subseteq I$, such that: $L_{q'_F}(D_G^{FR}) \subseteq \bigcup_{i \in J} L_{(q'_F, q_i)}(V_G)$ and $\bigcap_{i \in J} L(V_G, (q_F, q_i)) \neq \varnothing$. This equivalence provides an algorithm for checking non-strong manifestability, which boils down to a finite number of tests of language equivalence and language emptiness. In the worst case, this algorithm may require testing all subsets J of I, thus giving an EXPTIME complexity in the size of G. Nevertheless, under the particular assumption that there is no cycle in G before the occurrence of F or containing F, the system has then only a finite number of fault occurrences, i.e., of possible prefixes sF, as the language $L(D_G^{FR}, q_F)$ is finite. Processing each word sF of this language separately, one has just to do each time one language equivalence test between the fault refined diagnoser and the pair verifier limited to sF, which gives a PSPACE complexity for the corresponding algorithm. This proves that checking strong manifestability is a PSPACE-complete problem for the class of systems verifying this assumption.

5.2.3 . Experimental Results

We have theoretically proved the correctness of our Algorithm. To show its feasibility, we implemented Algorithm 1, including emptiness and equivalence checking, on a Mac OS laptop with a 1.7 GHz Intel Core i7 processor and 8 Go 1600 MHz DDR3 memory, and applied it on more than one hundred examples, including literature and hand-crafted ones. The latter were built to show the scalability, as the former are very small in size.

LitSys	$\|S\|/\|T\|$	$\|S\|/\|T\|(PV)$	Time	Verdict	HCSys	$\|S\|/\|T\|$	$\|S\|/\|T\|(PV)$	Time	Verdict
Ex.12	8/10	4/4	15	SManifes	h-c1	22/24	18/18	32	SManifes
[85]	16/23	7/11	39	Manifes	h-c2	36/39	74/77	90	Manifes
[73]	16/20	7/9	25	Manifes	h-c3	87/90	63/68	105	Manifes
[54]	4/7	3/3	12	SManifes	h-c4	52/57	32/30	63	SManifes
[100]	15/21	11/16	52	SManifes	h-c5	57/69	32/37	78	SManifes
[84]	11/15	2/1	16	Diagno	h-c6	509/570	79/81	132	Manifes
[82]	8/12	8/11	53	NManifes	h-c7	986/1032	870/861	312	NManifes

Table 5.1: Experimental results of manifestability checking for DESs

Some of our experimental results are shown in Table 5.1. On the left of the table are examples (LitSys) including our Example 12 with illustrative examples of other papers. On the right are hand-crafted examples (HCSys) constructed by extending the examples (LitSys), while focusing on non-diagnosable examples. For example, for a manifestable system, an arbitrary automaton without fault is added

such that at least one faulty trajectory can always manifest itself (and obviously critical pairs are preserved as nothing is deleted from the system model). We give the number of states and transitions of the system ($|S|/|T|$), of the pair verifier ($|S|/|T|(PV)$), as well as the execution time (in milliseconds) and the verdicts including Manifes(tability), Diagno(sability), N(on-)Manifes(tability), which show the strongest property satisfied by the system, strong level described by Theorem 7.

From our experimental results, it appears that the algorithm execution time depends on the size of the system and the size of the pair verifier. We can see that the original HVAC system in [82] (as well as its extension h-c7) is not manifestable. It is thus necessary to go back to design stage to revise the system model. Also, we employ the construction in the proof of Theorem 11 as hand-crafted example h-c7 to achieve a worst case. Now, for manifestable but not diagnosable systems, it is more interesting to study time bounded-manifestability by taking time constraints into account, making sure to detect the fault in bounded time after its occurrence. This is the subject of the next section.

5.3 . Manifestability Analysis for Real-Time Systems

In the previous section, we proposed a new system property called manifestability and designed an algorithm to check manifestability for discrete-event systems whose complexity is PSPACE. In this section, we extend our approach to real-time systems modeled by timed automata. To do so, we redefine manifestability by taking into account time constraints. We prove that the problem is undecidable for general timed automata but we define a particular subclass of them for which it is PSPACE.

5.3.1 . Motivation/Introduction

In real life, there are many complex time-dependent systems, i.e., real-time systems, where explicit time constraints must be taken into account when analyzing the behavior of the system. These time constraints naturally exist in real-life systems, such as transmission delays, response times, etc., and therefore cannot be neglected considering their impact on some properties, such as manifestability. For example, in an untimed DES, if there are two ambiguous behaviors, by adding explicit time constraints, e.g., the delay between some two successive observable events is bounded, we can easily distinguish them. However, such time constraints cannot be expressed in classical models such as automata and Petri nets, so we choose to adopt timed automata (TA) as real-time system models to analyze manifestability thanks to their formal semantics. In timed automata, quantitative properties of delays between events can easily be expressed. The execution traces of TA are modeled by timed words, i.e., sequences of events associated with the time at which they occur. TA are thus considered as acceptors for the languages of

timed words. In this section, we thus extend our approach to TA for manifestability problem.

5.3.2 . Manifestability Property for RTSs

As seen in section 2.2.2, timed automata constitute a framework for modeling and verifying RTSs. We thus redefine manifestability for timed automata.

Definition 24 (Manifestability of TA) *F is manifestable in a TA A if*

$$\exists \rho \in L_F(A),$$
$$\forall \rho' \in L(A), P\left(\rho'\right) = P(\rho) \Rightarrow F \in \rho'.$$

Note that we could also adopt a weaker definition of manifestability that allows ρ to be an arbitrary time-finite run, i.e., not only a finite run in $L_F(A)$ but also a Zeno run in $L_F^{\omega}(A)$, but as Zeno runs are usually non-desirable behaviors due to modeling errors, we adopt this stronger version excluding manifestability via Zeno runs only. By using Definition 9, a straightforward rephrasing of Definition 24 gives the following result, which is similar to Theorem 5.

Theorem 12 *A fault F is manifestable in a TA A iff the following condition is satisfied (where $\not\approx_{\Delta}$ denotes a timed critical pair, see Definition 9):*

$$(\mathcal{M}^t) \quad \exists \rho \in L_F(A), \nexists \rho' \in L_N(A), \rho \not\approx_{time(\rho,F)} \rho'.$$

In the same way, Definition 19 can be adapted to define strong manifestability of TA.

Definition 25 (Strong Manifestability of TA) *Given a TA A and a fault F:*

1. *given $\Delta \in R, F$ is strongly Δ-manifestable in A if*

$$\forall \rho^F \in L(A), \exists t \in L(A)/\rho^F, \mathrm{time}(t) \leqslant \Delta,$$
$$\forall \rho' \in L(A), P(\rho') = P(\rho^F t) \Rightarrow F \in \rho'.$$

2. *F is strongly manifestable in A if*

$$\exists \Delta \in R \text{ such that } F \text{ is strongly } \Delta\text{-manifestable in } A.$$

Similar to Theorem 6.1, we obtain the following result.

Theorem 13 *Given a TA A, a fault F and $\Delta \in R, F$ is strongly Δ-manifestable in A iff the following condition is satisfied:*

$$\left(\mathcal{M}_{\Delta}^{ts}\right) \quad \forall \rho^F \in L(A), \exists t \in L(A)/\rho^F, \text{ time } (t) \leqslant \Delta,$$
$$\nexists \rho' \in L_N(A), \rho^F t \not\approx_{\text{time } (t)} \rho'.$$

Therefore, in a similar way as for DESs, verifying manifestability for TA consists in checking the existence of a faulty trajectory that can be distinguishable from all normal ones. But in timed automata, we must consider the occurrence time of observable events, i.e., a non-manifestable DES possibly becomes manifestable by adding some time constraints such that at least one faulty trajectory can be distinguishable from normal ones thanks to the different occurrence time of the same observable events.

Example 13 *Consider the system modeled by the TA in Figure 5.6. If there were no time constraints, that is considering just the underlying automaton modeling a simple DES, we can see it would not be manifestable since all faulty trajectories would have the same observations as the normal ones, i.e.,* $o1o2o3^*$*. With the addition of time constraints as shown on the figure, at least one faulty timed trajectory can manifest itself and be distinguished from the normal one, in such a way that the system, now a timed automaton, becomes manifestable. More precisely, the faulty trajectory with the event* $u1$ *can be distinguished from the normal one since the time duration between the successive observable events* $o1$ *and* $o2$ *is not larger than 3 time units for the former, whereas it is larger than 3 time units for the latter. When the time elapsed between the observations of* $o1$ *and* $o2$ *is not larger than 3, we can be sure a fault has occurred.*

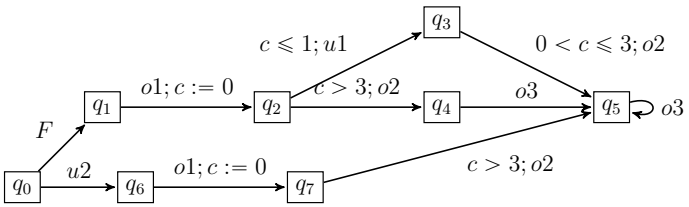

Figure 5.6: A real-time system model TA.

It is therefore clear that the addition of time constraints can sometimes turn a non-manifestable system into a manifesatble one by distinguishing temporally a faulty trajectory, which cannot manifest itself in the untimed setting, from all normal trajectories.

5.3.3 . Undecidability and Decidability Results

From a given TA A modeling a real-time system, we first construct its corresponding fault diagnoser D_A^F as Definition 21 and refined normal diagnoser D_A^{NR} as Definition 22, then synchronise D_A^F with D_A^{NR} based on the set of observable events to obtain the fault pair verifier V_A^F as Definition 23 (it is not necessary to do the refinement of D_A^F; the reason for limiting as much as possible the use of the refinement process for timed automata is explained just below). We also

define the final states in D_A^F and V_A^F as the faulty states and the ambiguous states, respectively. Thus, manifestability verification consists in checking whether there does exist an accepted timed trajectory in D_A^F that is not accepted by V_A^F. The reason is that each ambiguous timed trajectory in V_A^F corresponds to a faulty timed trajectory in the original system, for which there exists at least one normal timed trajectory with the same observation, i.e., such that the fault cannot manifest itself.

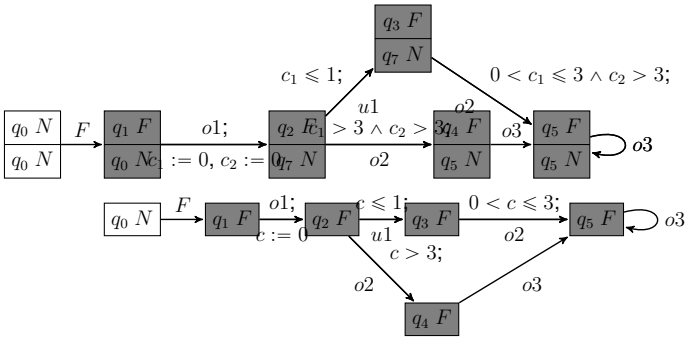

Figure 5.7: The fault pair verifier V_A^F for the system whose model is depicted in Figure 4.1 (top); the fault diagnoser D_A^F (bottom).

Example 14 *For the example depicted in Figure 5.6, V_A^F and D_A^F are shown in Figure 5.7. Since we synchronize two timed trajectories in V_A^F, their corresponding clock variable c is distinguished by renaming as c_1 and c_2 [94]. Since the transition in V_A^F following $u1$ can never be fired due to its clock constraints, any timed trajectory of D_A^F containing unobservable event $u1$ (and $o3$) is not accepted by V_A^F, which proves that F is manifestable.*

The problem in the general case is that, to construct D_A^{NR} from D_A^N , we are obliged to rest on the delay closure operation, i.e., on removing unobservable events or equivalently removing ϵ-transitions. But it is known that this is not always possible. Actually, it has been proved [17] that, contrary to the case of FSM, ϵ-transitions strictly increase the power of TA, if there is a self-loop containing ϵ-transitions which reset some clocks. However, ϵ-transitions can be removed if they do not reset clocks, in order to obtain a TA accepting the same timed language. More generally, it has been proved [44] that a TA such that no ϵ-transition with non-empty reset set lies on any directed cycle can be effectively transformed into a TA without ϵ-transitions that accepts at least the timed language of the initial TA and whose non-Zeno accepted timed words are the same with those accepted by the initial TA. In our study, since we do not exclude Zeno runs, we will assume simply that in the normal diagnoser D_A^N, there is no clock reset for the transitions with unobservable events. Other unobservable events are not handled as ϵ-transitions.

Example 13 fulfills this assumption since clock c is not reset in transition labeled $u2$ (but it could be reset in transition labeled $u1$, which does not exist in D_A^N). Thus, in our case, we adopt the method proposed in [17] to obtain D_A^{NR} by removing unobservable events in D_A^N.

Assumption 5 (Limited clock reset TA) *Any transition in A from a state q reachable from q_0 by a normal (i.e., not containing F) execution, and associated to an unobservable normal event σ, is of the form $(q, g, \sigma, \varnothing, q')$, i.e., has no clock reset.*

Assumption 5 formally states that in our study there is no clock reset for transitions with unobservable events in D_A^N. Thus, we can further extend Theorem 9 to TA from the construction of the two structures D_A^F and D_A^{NR}.

Theorem 14 *Given a real-time system model A with limited clock reset, a fault F is manifestable in A iff $L_a\left(V_A^F\right) \subset L_F\left(D_A^F\right)$, with $V_A^F = D_A^F \|_{\Sigma_o} D_A^{NR}$.*

Now, we can check manifestability as checking inclusion between the languages defined by the two TA V_A^F and D_A^F. However, it is well-known that this problem is undecidable for general TA [5]. Actually, in a similar way to that in the discrete framework, we can reduce the inclusion problem of TA to the manifestability problem of TA, which proves that checking manifestability in TA is undecidable.

Theorem 15 *Given a TA A and a fault F, the problem of checking whether F is manifestable in A is undecidable.*

Proof 7 *Reducing the undecidable inclusion problem of TA to the manifestability problem is achieved by adapting to TA the construction in the proof of Theorem 11. Let $A_1 = (Q_1, \Sigma, X_1, \delta_1^{X_1}, q_0^1, I_1)$, $A_2 = (Q_2, \Sigma, X_2, \delta_2^{X_2}, q_0^2, I_2)$ be two arbitrary (non-deterministic) time living TA on the same vocabulary defining prefix-closed timed languages. One can assume that $Q_1 \cap Q_2 = \varnothing$. Based on A_1 and A_2, one can construct a new TA representing a system model, $A = (Q, \Sigma \cup \{\tau, F\}, X, \delta^X, q_0, I)$, where $Q = Q_1 \cup Q_2 \cup \{q_0\}$, $X = X_1 \cup X_2 \cup \{x_0\}$, $\delta^X = \delta_1^{X_1} \cup \delta_2^{X_2} \cup \{(q_0, x_0 = 0, F, \varnothing, q_0^1), (q_0, x_0 = 0, \tau, \varnothing, q_0^2)\}$ and $I = I_1 \cup I_2$, with $\Sigma_o = \Sigma$, $\Sigma_u = \{\tau\}$ and $\Sigma_f = \{F\}$. A satisfies the assumption of limited clock reset. From the construction of A, one has $L(A_1) = P(L_F(A))$ and $L(A_2) = P(L_N(A))$. In the same way as the proof of Theorem 11, one gets finally $L(A_1) \cap L(A_2) \subset L(A_1) \Longleftrightarrow F$ is manifestable in A, i.e., $L(A_1) \subseteq L(A_2) \Longleftrightarrow F$ is not manifestable in A. So, languages inclusion testing for TA boils down to manifestability checking of TA. The proof shows also that checking strong manifestability is undecidable.*

Since the manifestability problem of TA is undecidable, we now analyze a subclass of TA whose manifestability problem is decidable. The idea comes from the fact that the inclusion problem of deterministic TA is PSPACE-complete [5]. Where, for a TA A, we say it is deterministic whenever given two distinct discrete transitions from the same state with the same label $(q, g_1, \sigma, r_1, q'_1)$ and $(q, g_2, \sigma, r_2, q'_2)$, it holds that $g_1 \wedge g_2$ is unsatisfiable, i.e., there is no common time where one or the other could be indifferently fired.

Definition 26 (Single-Silent Deterministic TA) *Given a TA A with limited clock reset, we call it Single-Silent Deterministic TA (SS-DTA), if A is deterministic and, from any state q reachable from q_0 by a normal execution, if a transition by an unobservable normal event exists from q, then it is the only one normal transition from q, i.e., if $(q, g_1, \sigma_1, r_1, q'_1)$ and $(q, g_2, \sigma_2, r_2, q'_2)$ are two different transitions with $\sigma_1, \sigma_2 \neq F$, then $\sigma_1 \in \Sigma_o$ and $\sigma_2 \in \Sigma_o$.*

The TA of Figure 5.6 is an SS-DTA. We can see that for an SS-DTA A, as A is deterministic, so are both its normal diagnoser D_A^N and its fault diagnoser D_A^F. The condition of Definition 26 also implies that any unobservable normal transition in D_A^N is the only one transition issued from its source state. And with Assumption 5, it is easy to show that D_A^{NR} (obtained by deleting unobservable events in D_A^N) keeps deterministic. Since both D_A^{NR} and D_A^F are deterministic, then V_A^F obtained from their synchronization is also deterministic (this can be verified for the example in Figure 5.7).

Theorem 16 *Given an SS-DTA and a fault F, its manifestability problem is PSPACE-complete.*

By adopting a similar way as for the proof of Theorem 11, Theorem 16 can be proved by reducing the inclusion problem of two deterministic TA to manifestability problem of an SS-DTA (note that Assumption 5 and the second condition of Definition 26 are trivially satisfied as τ is the unique unobservable normal event).

5.3.4 . Encoding Bounded Manifestability

In this section, we will show how to verify the manifestability property of a SS-DTA by encoding it into SMT formula. As it is already known that the problem of deciding whether a TA is determinizable is actually undecidable [47], we do not consider the problem of deciding if an arbitrary TA can be transformed into a SS-DTA. However, there exist some subclasses of TA that are determinizable by using for example the algorithm proposed in [10]. To make easier SMT encoding for the inclusion checking problem, we add to the deterministic fault pair verifier V_A^F an additional non-final state *sink*, such that V_A^F is deterministic and complete. This is done by adding transitions from other states to the state *sink* and a self-loop for

state *sink* (see [4] for details, in the case where invariants are $True$). By Theorem 14, checking manifestability of an SS-DTA A is equivalent to finding a faulty timed trajectory ρ, which is accepted by D_A^F, such that the timed trajectory of V_A^F identical to ρ (which exists as V_A^F is complete and is unique as V_A^F is deterministic) is not accepted by V_A^F. Thus we can check manifestability via finding a timed trajectory which is accepted by D_A^F and rejected by V_A^F. To code this problem as a (finite) logical formula whose satisfiability will be determined by bounded model checking, it is necessary to bound the length of the timed trajectories considered. Thus given an input integer parameter k, only timed trajectories ρ such that $|\rho| \leqslant k$ will be considered. Now, for the end-user, it is important that, in case of manifestability, the fault will manifest itself after an acceptably long time. That is, his requirement will not be the length k, but the time delay $\triangle$ indicating a time upper-bound after its occurrence for the fault to manifest itself (similar to the concept of $\triangle$-manifestability used in Definition 25.1). In general, there is no relationship between k and $\triangle$ among timed trajectories (except that, for a given timed trajectory, the two parameters vary in the same sense), as a longer timed trajectory may have a smaller time, thus the usage of the method is as follows: one checks if it exists a timed trajectory of length at most k (input of the algorithm) and of time after fault at most $\triangle$ (input from the end-user), in which case the requirements of the end-user are realized; otherwise one repeats the process with a greater k. One can at any step query without the parameter $\triangle$ in order to get back a delay $\triangle'$ (greater than $\triangle$), proving the manifestability in time $\triangle'$ after the fault, and see if it could be acceptable by the end-user. Obviously, if no theoretical length upper-bound exists for manifestability, the absence of solution will not be a guarantee that the fault is not manifestable.

Encoding (deterministic) TA

We now show how to logically encode bounded manifestability in SMT, i.e., checking the existence of a timed trajectory (of length at most k and postibly of time after fault at most $\triangle$) accepted by D_A^F and rejected by V_A^F, such that the satisifiablity of the logical formula is equivalent to the existence of such a trajectory, which witnesses manifestability of the system. If the result is satisfiable, a model is returned, which actually provides such a timed trajectory. As shown in Section 2.2.2, we can assume that time and discrete transitions alternate in any timed trajectory. Hence, we rewrite $(q, v) \xrightarrow{t} (q, v'') \xrightarrow{\sigma} (q', v')$, where $t \in R$ and $\sigma \in \Sigma$, as $(q, v) \xrightarrow{t,\sigma} (q', v')$. We consider this kind of combined time-discrete transition during the SMT encoding. Accordingly, a timed trajectory of length k is a finite sequence $(t_0, \sigma_0), (t_1, \sigma_1), \ldots, (t_{k-1}, \sigma_{k-1})$, where $t_i \in R, \sigma_i \in \Sigma$, and $\forall i, 0 \leqslant i \leqslant k-1, (q_i, v_i) \xrightarrow{t_i,\sigma_i} (q_{i+1}, v_{i+1})$ is allowed by A. We can assume that the timed trajectory ends by a time transition, that we will represent by setting $\sigma_{k-1} = \epsilon$ as a silent event.

Example 15 *For the example depicted in Figure 5.6, one 4-length timed trajectory is* $\rho = (1.5, u2), (3, o1), (5, o2), (1, \epsilon)$ *that is witnessed by the feasible execution* $(q_0, c = 0) \xrightarrow{1.5,u2} (q_6, c = 1.5) \xrightarrow{3,o1} (q_7, c = 0) \xrightarrow{5,o2} (q_5, c = 5) \xrightarrow{1,\epsilon} (q_5, c = 6)$.

Given a TA A and a fault F, to check its manifestability, we first construct the fault pair verifier V_A^F and the fault diagnoser D_A^F from A as described before. We denote them $V_A^F = \left(\hat{Q}, \Sigma, \hat{X}, \hat{\delta}^X, \hat{q}_0, \hat{I}\right)$ and $D_A^F = \left(Q, \Sigma, X, \delta^X, q_0, I\right)$, both with Σ, the set of events of A. Then similar to Section 2.5.5, we encode the essential static parts in V_A^F and D_A^F as follows.

- The set of states is encoded by positive integers with the function $E_Q : Q \rightarrow Q^E = \{1, ..., \|Q\|\}$ (resp., $\hat{E_Q} : \hat{Q} \rightarrow \hat{Q}^E = \{1, ..., \|\hat{Q}\|\}$, where $Q^F \subseteq Q^E$ (resp. $\hat{Q}^F \subseteq \hat{Q}^E$) codes the final states, i.e., Q^F corresponds to the set of faulty states (resp., $\hat{Q}^F$ to the set of ambiguous states).

- The set of events for both TA is encoded by positive integers $E_\Sigma : \Sigma \rightarrow \Sigma^E = \{1, ..., \|\Sigma\|\}$, where $\Sigma^E = \Sigma_o^E \uplus \Sigma_u^E \uplus \Sigma_f^E$, corresponding to $\Sigma = \Sigma_o \uplus \Sigma_u \uplus \Sigma_f$. The normal events $\Sigma_n = \Sigma_o \uplus \Sigma_u$ are encoded by integers from 1 to $\|\Sigma_n\|$ and fault events by integers from $\|\Sigma_n\| + 1$ to $\|\Sigma\|$.

- The set of symbolic transitions is encoded by a set of tuples $E_{\delta^X} : \delta^X \rightarrow \delta^E = (Q^E \times \mathbb{C}(X) \times \Sigma^E \times 2^X \times Q^E)$ with $E_{\delta^X}(q, g, \sigma, r, q') = (E_Q(q), g, E_\Sigma(\sigma), r, E_Q(q'))$. A similar way to define $\hat{E}_{\delta^X}$ on $\hat{\delta}^X$ for $\hat{\delta}^E$.

Encoding Bounded Manifestability

Given k and Δ, the essential point is to define a formula Ψ_Δ^k whose satisfiability is equivalent to the existence of a timed trajectory ρ with $|\rho| = k$ and time$(\rho, F) \leqslant \Delta$ which is accepted by D_A^F and rejected by V_A^F. In order to describe the formula Ψ_Δ^k as intuitively as possible, we present it with different separate parts. The variables used in the encoding of the timed trajectory are similar to those presented in 2.5.6: those equipped with a hat are associated to V_A^F while those without a hat are attached to D_A^F, except for variables representing the events and the time periods which are now the same, say, without a hat (and there is no need here of global clock).

- Initialization. The two timed trajectories should start in the initial state with the initialization of all clock variables. We set initial state as q_0, initial clock value as 0.

  - For the timed trajectory in D_A^F :

$$\Phi^{Init} := (\bigwedge_{x \in X} v_0^x = 0) \wedge (s_0 = E_Q(q_0)) \wedge (v_0^F = -1).$$

  - For the timed trajectory in V_A^F :

$$\hat{\Phi}^{Init} := (\bigwedge_{x \in \hat{X}} \hat{v}_0^x = 0) \wedge (\hat{s}_0 = \hat{E}_Q(\hat{q}_0)).$$

- Well-formedness of timed trajectories. Three points have to be verified for well-formedness: 1) each time period between two discrete transitions should be non-negative; 2) the values of integer-valued variables representing all events should be in $\{1 \dots \|\Sigma\|\}$; 3) the values of variables representing all states should be in $\{1 \dots \|Q\|\}$ for D_A^F and in $\{1 \dots \|\hat{Q}\|\}$ for V_A^F. As it is about the same timed word, it is enough to check the first two points only once.

  - For the timed trajectory in D_A^F :

$$\Phi^{WF} := (\bigwedge_{i=0}^{k-1} 0 \leqslant t_i) \wedge (\bigwedge_{i=0}^{k-1} 1 \leqslant e_i \wedge e_i \leqslant \|\Sigma\|) \wedge (\bigwedge_{i=0}^{k-1} 1 \leqslant s_i \wedge s_i \leqslant \|Q\|).$$

  - For the timed trajectory in V_A^F :

$$\hat{\Phi}^{WF} := (\bigwedge_{i=0}^{k-1} 1 \leqslant \hat{s}_i \wedge \hat{s}_i \leqslant \|\hat{Q}\|).$$

- Acceptance of the timed trajectory in D_A^F and rejection of the same timed trajectory in V_A^F. We formalize here that the timed trajectory represented by values for the predefined variables without hat should be accepted by D_A^F, where final states are faulty ones. And the timed trajectory represented by those for variables with hat should be rejected by V_A^F, where final states are ambiguous ones. Precisely, in each timed trajectory, each pair of adjacent states has to be connected by a transition that is allowed in the corresponding TA. The last state in the trajectory in V_A^F is not a final one, while the last state in D_A^F is a final one with the length bound k.

  - For the timed trajectory in D_A^F:

$$\Phi^{Acc} := (\bigwedge_{i=0}^{k-1} (\bigvee_{(s_i,g,e_i,r,s_{i+1}) \in \delta^E} [[g]]_i \wedge TP_i^r)) \wedge (\bigvee_{q \in Q^F} \hat{s}_k = q).$$

    Here $[[g]]_i$ represents that the clock valuations after the i-th step in this timed trajectory, i.e., $v_i^x + t_i$, should satisfy the guard g, such as:

    * $[[x \bowtie c]]_i := (v_i^x + t_i) \bowtie c.$
    * $[[x - y \bowtie c]]_i := (v_i^x - v_i^y) \bowtie c.$
    * $[[g_1 \wedge g_2]]_i := [[g_1]]_i \wedge [[g_2]]_i .$

    TP_i^r in the above expression formalizes the time progression, i.e., time transition, by resetting clocks in the subset r and by increasing all other clocks, including the time elapsed from the first fault occurrence if triggered, with the corresponding period t_i:

$$TP_i^r := (\bigwedge_{x\in r} v_{i+1}^x = 0) \wedge (\bigwedge_{x\in(X\setminus r)} v_{i+1}^x = v_i^x + t_i)$$
$$\wedge(0 \leqslant v_i^F \Rightarrow v_{i+1}^F = v_i^F + t_i).$$

– For the timed trajectory in V_A^F :

$$\hat{\Phi}^{Rej} := (\bigwedge_{i=0}^{k-1}(\bigvee_{(\hat{s}_i,\hat{g},e_i,\hat{r},\hat{s}_{i+1})\in\hat{\delta}^E} [[\hat{g}]]_i \wedge \widehat{TP}_i^r)) \wedge (\bigwedge_{q\in\hat{Q}^F} \hat{s}_k \neq q).$$

In a similar way, $[[\hat{g}]]_i$ for the timed trajectory in V_A^F is encoded as follows:

* $[[\widehat{x \bowtie c}]]_i := (\hat{v}_i^x + t_i) \bowtie c.$
* $[[\widehat{x - y \bowtie c}]]_i := (\hat{v}_i^x - \hat{v}_i^y) \bowtie c.$
* $[[\widehat{g_1 \wedge g_2}]]_i := [[\hat{g_1}]]_i \wedge [[\hat{g_2}]]_i.$

The following is the time progression for this timed trajectory:

$$\widehat{TP}_i^{\hat{r}} := (\bigwedge_{x\in\hat{r}} \hat{v}_{i+1}^x = 0) \wedge (\bigwedge_{x\in(X\setminus\hat{r})} \hat{v}_{i+1}^x = \hat{v}_i^x + t_i).$$

- The timed trajectory contains a fault occurrence (with one fault type, the fault occurrence coding can be simplified as $\|\Sigma_n\| + 1 = e_i$). Furthermore, after the first occurrence of a fault at step i, the value of the variable v_{i+1}^F is assigned to 0 to trigger counting the time elapsed from this fault occurrence (otherwise it stays equal to -1). Finally, we check whether the time elapsed after fault is at most Δ (in absence of given Δ, nothing is added).

$$\Phi^{\Delta} := (\bigwedge_{i=0}^{k-1}(v_i^F = -1 \Rightarrow ((\|\Sigma_n\| < e_i \Rightarrow v_{i+1}^F = 0)$$
$$\wedge(e_i \leqslant \|\Sigma_n\| \Rightarrow v_{i+1}^F = -1)))) \wedge v_k^F \leqslant \Delta.$$

Now the formula Ψ_{Δ}^k whose satisfiability witnesses manifestability (in time after fault at most Δ) is presented as follows:

$$\Psi_{\Delta}^k := \Phi^{Init} \wedge \hat{\Phi}^{Init} \wedge \Phi^{WF} \wedge \hat{\Phi}^{WF} \wedge \Phi^{Acc} \wedge \hat{\Phi}^{Rej} \wedge \Phi^{\Delta}.$$

Note that for the sake of simplicity, in the proposed formula, there is no state invariant. But considering timed automata without state invariants does not entail any loss of generality as the invariants can be added to the guards [56]. And, if really wanted, the formula Ψ_{Δ}^k can be extended to handle such invariants (by verifying that the clock valuations in each state do not violate the corresponding invariant, which has to be done only when entering the state and leaving it).

5.3.5 . Experimental Results

In this section, we provide some experimental results to show the feasibility of our approach to check manifestability of TA. We realized a prototype implementation in Python by using the SMT solver Z3. The program was executed on the same machine as for the first set of experiences shown in Section 5.2.3.

Given an SS-DTA A, we construct its fault pair verifier V_A^F as described in Section 5.3.3, which is done at the syntactic level. Then, based on D_A^F and V_A^F, we encode the formula Ψ_Δ^k as described in Section 5.3.4. The satisfiability of Ψ_Δ^k, i.e., the construction of a corresponding adequate timed trajectory, is checked by Z3. With a bounded model checking process, we test for different values of the bound k (length of the trajectory and thus measure of the size of the formula). We report on different versions of three literature examples, including Example 13 which is ex_{00}, that are modified by adding different temporal constraints such that we have both manifestable and non-manifestable models for each of them. Note that some original literature examples are finite state automata. For example, ex_{01} is obtained from ex_{00} by changing the guard from q_3 to q_5 as $c \geqslant 3$ and becomes thus non-manifestable because no faulty timed trajectory can manifest itself. Furthermore, considering that such literature examples are normally quite small, to show the scalability, we have tested also some hand-crafted systems (hcs), constructed in a partially random way based on the chosen literature ones without changing the verdict. For example, ex_{02} is constructed based on ex_{00} by adding a deterministic TA whose initial state is the destination state of an additional transition with source state q_2, remaining thus manifestable. Similarly, ex_{03} is generated from ex_{01} by adding a deterministic TA without fault to the state q_6, and remains thus non-manifestable.

Table 5.2 shows part of our experimental results, where column 2 shows the number of transitions of the corresponding system model, columns 3 and 4 the upper bound k for the length of timed trajectories and the time upper bound Δ after fault occurrence. Then one can find the size of the formula expressed by its number of clauses, the required memory and the execution time in seconds in the columns 5, 6 and 7, respectively. The final column shows the verdict for each system, where *SAT* witnesses manifestability, while *UNSAT* implies non-manifestability. For the manifestable systems, we try to give k and Δ as small as possible. A small Δ is interesting from a practical point of view since it represents how much time after the fault occurrence this fault manifests itself. Another important observation is that for non-manifestable systems, we increase the value of k as well as Δ to show the scalability. From the size of the formulas, one can see that SMT solvers can check for satisfiability relatively large formulas.

Sys	$\|trans.\|$	k	Δ	$\|clauses\|$	$mem.$	time	$SAT?$
ex_{00}	10	5	1	668	3.98	0.09	SAT
ex_{01}	10	430	10000	89423	18.28	859.72	UNSAT
ex_{02}(hcs)	233	5	3	16781	8.76	1.78	SAT
ex_{03}(hcs)	365	51	10000	510972	17.50	802.53	UNSAT
ex_{04}(hcs)	1782	7	5	441563	29.27	573.36	SAT
ex_{05}(hcs)	1620	15	1000	512308	30.22	1216.82	UNSAT
ex_{10} [54]	6	2	3	243	2.50	0.03	SAT
ex_{11} [54]	6	420	10000	118267	16.62	767.73	UNSAT
ex_{12}(hcs)	287	3	5	6051	5.31	0.63	SAT
ex_{13}(hcs)	381	56	20000	557073	18.21	721.15	UNSAT
ex_{14}(hcs)	2030	7	5	36754	25.09	37.81	SAT
ex_{15}(hcs)	2436	9	20000	521857	32.63	763.02	UNSAT
ex_{20} [59]	9	5	1	578	3.6	0.12	SAT
ex_{21} [59]	9	380	15000	127778	17.60	743.43	UNSAT
ex_{22}(hcs)	296	5	2	16008	5.52	2.1	SAT
ex_{23}(hcs)	315	32	20000	507318	17.53	933.21	UNSAT
ex_{24}(hcs)	2120	5	3	120003	27.09	90.06	SAT
ex_{25}(hcs)	1695	8	23000	423305	28.36	1545.20	UNSAT

Table 5.2: Experimental results of manifestability checking for SS-DTA

5.4 . Related Work

In [86, 87], the authors proposed different variants of detectability (such as strong detectability) about state estimation. The system is detectable (resp., strongly detectable) if, based on a sequence of observations, one can be sure about the state in which the system is for some given trajectory (resp., all trajectories). They proposed a polynomial algorithm for strong detectability, for which two different trajectories with the same observations witness its violation. However, to analyze detectability, they constructed a deterministic observer that has exponential complexity with the number of system states. Our approach can be adapted to handle state estimation by considering an ambiguous state as one that contains different system states. Thus, we can improve their state estimation by using the improved equivalence checking techniques (e.g., the approach of [25] normally constructs a small part of the deterministic automaton). Furthermore, we proved that the problem of manifestability itself is PSPACE-complete.

The authors of [1, 50] proposed an approach for weak diagnosability in a concurrent system by using Petri nets, i.e., impose a constraint of weak fairness by disallowing the enabled transition to be perpetually ignored. The idea is to make impossible some non-diagnosable scenarios in order to upgrade the diagnosability level. They focused on how to get a more appropriate model, based on which a polynomial solution like that for classical diagnosability can be applied.

Two definitions for stochastic diagnosability were introduced and analyzed in [93], which are weaker than diagnosability. A-diagnosability requires that the

ambiguous behaviors have a null probability. While AA-diagnosability admits errors in the provided information which should have an arbitrary small probability. Then four variants of diagnosability (FA, IA, FF, IF) were introduced and studied for different probabilistic system models [19, 20]. Different ambiguity criteria were then defined according to different types of runs: for faulty runs only or for all runs; for infinite runs or for finite sub-runs. Among them IF-diagnosability (for infinite faulty runs) is the weakest one. Note that IF-diagnosability of a finite probabilistic system is equivalent to A-diagnosability.

The authors of [54, 18] analyzed (safe) active diagnosability by introducing controllable actions for (probabilistic) DESs, where the complexity of these problems was also studied. The idea is to design controllers (resp., label activation strategies for probabilistic version) to enable a subset of actions in order to make the system diagnosable (resp., stochastically diagnosable).

On the other hand, concerning TA, [26] analyzed the diagnosability problem of TA by constraining the class of diagnosers considered and demonstrated that it is 2EXPTIME-complete for a deterministic TA diagnoser, by using timed game construction. Some works proposed to use SMT techniques to perform verification on TA with quite good results. In [8], a SMT-based approach was proposed to incrementally analyze TA for some special decidable problems, including universality for deterministic TA and language inclusion of a non-deterministic one into a deterministic one. This is done by adopting bounded version for the sake of efficiency. To verify reachability for TA, [61] introduced a SMT-based bounded model checking to handle non-lasso-shaped infinite runs by integrating region abstraction. More recently, attention was payed to verification of special failure models, called Failure Propagation Models (FPMs), where failure propagation information is abstracted from the original system model. The approach proposed in [27] presents how to encode in SMT the diagnosability problem for a given timed FPM. It is worth noting that TA are totally different from FPMs, the former being considered as original system models, based on which FPMs can be abstracted. However, this transformation is not trivial at all, as demonstrated in ([22, 23]. Then, we have proposed in [55] a new approach to verify diagnosability directly on TA by using SMT techniques, which provides an alternative to systems for which the abstraction to a FPM is not convenient.

5.5 . Comparison with Opacity

A very close research field worth comparing in this dedicated section is the opacity analysis of discrete event systems, introduced in 2005, which has become a very fertile field of research over the last decade, driven by safety and privacy concerns in network communications and online services (see [58] for a survey). A system is opaque if an external observer (the intruder) is unable to infer a "secret" about the system behavior, i.e., if for any secret behavior, there exists at least one

other non-secret behavior that looks the same (for observation) to the intruder. In our context, if we consider the occurrence of a fault as the secret, and thus faulty trajectories as secret behavior and normal trajectories as non-secret behavior, then intuitively fault manifestability and opacity are dual concepts, each one being in some sense the negation of the other. But, as there are various notions of opacity and as fault occurrence is a specific type of secret, the various concepts and their relationships have to be studied carefully.

For DESs, opacity properties are classified into two families: language-based opacity (LBO) and state-based opacity (SBO), depending if a language or a set of states is the secret. The closest to fault manifestability is LBO, which is not surprising as it has been already shown in [66] that related properties such as observability, diagnosability and detectability can all be reformulated as opacity. Indeed, defining, for a system model G with fault F, the secret language L_S as $L_F(G)$ and the non-secret language L_{NS} as $L_N(G)$, then the strong opacity of L_S with respect to L_{NS} and P, defined in [66] as any word of L_S has same projection by P that some word of L_{NS}, is exactly equivalent to the negation of F manifestability. Actually, manifestability is directly related to a special case of opacity, called secrecy [9]. A language property of a system is said strongly secret if it is strongly opaque with respect to its complement. Considering to be faulty as property, i.e., considering as language the faulty trajectories, we obtain that strong secrecy is equivalent to the negation of manifestability. As checking strong secrecy has been proved to be PSPACE-complete [31], it results that checking manifestability is at most PSPACE (actually also PSPACE-complete as we proved, and it is the same for strong opacity).

A smoother LBO property, named weak opacity, is also defined in [66] as some word of L_S has same projection by P that some word of L_{NS}. And, analogously, weak secrecy for a property is defined as its weak opacity with respect to its complement (i.e., $L_{NS} = L(G) \backslash L_S$). It is proved in [105] that checking weak opacity is polynomial. But this concept of weak secrecy is not pertinent in the context of fault manifestability, as its negation would mean that any faulty trajectory is distinguishable from all normal trajectories, which never happens (any trajectory ending by a first occurrence of the fault cannot be distinguished from its normal longer strict prefix). Nevertheless, changing slightly the definition of L_S as faulty trajectories with at least one observable event after the fault occurrence, then the negation of weak secrecy would be exactly 1-step diagnosability, i.e., each occurrence of the fault is diagnosable from the first observation after its occurrence, which is a very strong property. Our strong manifestability is actually much more smooth, while having no studied equivalence in opacity. Indeed, the negation of strong manifestability means that it exists a trajectory s^F ended by the fault F such that any trajectory with prefix s^F remains secret with respect to normal trajectories and P (i.e., has same observation that some normal trajectory). Thus this particular secrecy does not concern any faulty trajectory as strong secrecy or some

faulty trajectory as would do weak secrecy, but any faulty trajectory having some given minimal faulty prefix. One points here a *specificity* of fault manifestability with respect to general secrecy or opacity properties, i.e., by construction the secret language L_S considered is suffix-closed in $L(G)$ (and thus L_{NS} is prefix-closed), expressing that the faults we consider are permanent.

Different in its approach, SBO, introduced by [78] for automata, relates to the intruder ability to infer that the secret is or has been in a given secret state or set of states. Depending on the nature of the secret set, different SBO properties have been defined [58]. Thus one can distinguish among others Current-State Opacity (CSO), if the intruder can never infer, from its observations, whether the current state of the system is a secret state or not (i.e., for every trajectory that leads to a secret state, there exists another trajectory with same observation leading to a non-secret state) and Initial-State Opacity (ISO), if the intruder is never sure whether the system's initial state was a secret state or not (i.e., for every trajectory that originates from a secret initial state, there exists another trajectory with same observation originating from a non-secret initial state). Both CSO and ISO have been proven to be PSPACE-complete and transformation mappings between LBO, CSO and ISO have been studied in [99]. *Note that our approach can be adapted by duality in a very straightforward way to analyze ISO, which can be considered as a special case of manifestability*: it is enough to add an initial state and transitions from this new initial state to the previous initial states, labeled with the fault event for those who are secret and with an unobservable normal event for those who are non-secret. However, the approach proposed in [81] to analyze ISO requires space complexity that is exponential in the number of states of the given automaton, which is hence improved by our method. Regarding CSO, we may have the following constatation: if we define secret states either as all states reachable by a faulty trajectory or those states that are destination states of a fault event, CSO does not apply to manifestability analysis (in particular, in CSO, a trajectory leading to a secret state may be normal).

In fact, most SBO properties are to mask the critical moments of the system, such that they cannot be revealed immediately to an external observer, and do not consider the system behavior once it has exited a secret state (in particular, the set of secret states is not required to be stable). Actually, the more general problem to keep secret the fact the system was in a secret state a few steps ago has been studied under the name of K-step opacity [78, 77], i.e., for every trajectory that leads to a secret state and every extension of it with at most K observable events, there exists another trajectory with same observation leading to a non-secret state with an extension with same observation that the previous extension (thus CSO is 0-step opacity). It has been proven to be NP-hard and was extended to infinite-step opacity [79, 77, 80, 102], proven to be PSPACE-hard. Note that here the goal is to mask a secret state by a non-secret state at the same place in the sequence of observations, which is insufficient in general to prevent an intruder for discovering

that a secret state was crossed at some place during the last K observations. To avoid this, a language-based translation of K-step opacity is suggested in [77] as trajectory-based K-step opacity, a stronger property ensuring that an intruder cannot determine whether the system has reached a secret state at any point during the last K observations (independently of its exact place). Actually, it looks to be identical to K-step strong opacity, introduced later in [46] to express that, for each trajectory, there exists a trajectory with same observation that never crossed a secret state during the last K observations. But again the dual notion, i.e., the presence of a secret sate in the last K observations necessarily manifests itself, is different from our strong k-manifestability, i.e., any fault event manifests itself in at least one of its future in at most k steps (could be as well k observations) after its occurrence. This is because our approach of fault manifestability, as fault diagnosability, is event-based and not state-based and thus the "faulty" character of a state is not related to that state but to the way it can be reached. In particular, a same state can be reached by a faulty trajectory and a normal one, i.e., a normal, so non-secret, trajectory may contain secret states. In a state-based framework of faulty systems, i.e., if a fault was characteristic of a state and possibly intermittent (i.e., the set of faulty states is not required to be stable), then there would exist a duality worthwhile to study between fault manifestability and SBO.

Opacity analysis for TA has been studied (almost exclusively) in [30], where the (language-based) opacity property for a secret timed language S with respect to a TA A and P is defined as the property that, for any run, it exists a run with same observation that does not belong to S. A state-based opacity property, called L-opacity is also defined, where a set SL of secret locations is said to be opaque with respect to A and P if, for any run, it exists a run with same observation whose last location reached does not belong to SL. L-opacity problem is proven to be undecidable, not only for general TA but also for DTA and even for the subclass of event-recording automata (ERA), where each clock is associated with an event and is reset when this event occurs. It is then shown that opacity can be reduced to L-opacity, with the consequence that opacity problem is undecidable even for ERA with secrets given by ERA. In the context of fault manifestability for a TA A, taking for S the language of faulty runs, we obtain that the opacity of S is equivalent to the negation of manifestability as we defined it. So, our undecidability result for checking manifestability of TA, for which we gave a direct proof, can be obtained by adapting the proof in [30]. In [98] the language-based opacity problem for real-time automata (RTA), a subclass of TA (not comparable with ERA) which has a single clock which is reset at each transition and thus can be regarded as finite state automata with time information for each transition, has been proven to be decidable without more precision. But, except this very particular subclass, there is no work, as far as we know, that succeeded to give a sufficient condition on TA such that the opacity problem becomes decidable. In our work, we proved that, for the subclass of SS-DTA, the manifestability problem is PSPACE-complete and we proposed an

SMT-based approach to check it. Another close property called non-interference is to guarantee the safety of flow information by capturing causal dependency between high-level actions (private) and low-level behavior (public). The authors of [49] analyzed different variants of this property for TA and proved some of them decidable by transforming them into weak simulation problem between TA with event set excluding private events and TA with that hiding private events.

5.6 . Conclusion

In this chapter, we have addressed the formal verification of manifestability for both DESs and real-time systems. To bring an alternative to (stochastic) diagnosability analysis, whose satisfaction is very demanding in terms of sensors placement, we have defined (strong) manifestability, a new weaker property, actually the weakest one to satisfy to have a chance to diagnose a given fault. It is especially useful when the stochastic model is not available during diagnosability analysis. Note that non-manifestability of a system implies its non-stochastic diagnosability, but the converse is not necessarily true. It is worth noting that for today's complex systems, it is not realistic to analyze (stochastic) diagnosability for each type of faults (e.g., hundreds of faults may occur for even one HVAC subsystem in a given building with different categories such as abrupt and degradation [65]). It is more reasonable to verify different properties (e.g., diagnosability for abrupt faults and manifestability for degradation faults) for different faults according to their severity. We also want to emphasize that if stochastic diagnosability is very useful and interesting when the fault occurrence probability distributions are available, very limited studies have been conducted about this availability even for quite mature HVAC systems [65].

We have demonstrated that manifestability problem for finite state automata (resp., TA) is PSPACE-complete (resp., undecidable). We further defined SS-DTA, a subclass of deterministic TA, for which this problem becomes PSPACE-complete. It is thus encoded into an SMT formula, which can be checked automatically by an SMT solver. The feasibility and reasonable scalability of this approach have also been shown by preliminary experimental results. With such tools at his disposal, the designer may thus check both manifestability and diagnosability of each given fault. If manifestability is not satisfied, he knows that this fault, if it occurs, will never be detectable and he has thus necessarily to add sensors to make it manifest itself. If the fault has been proven manifestable but non-diagnosable, he knows, from the outputs of the algorithms, both a future trajectory of the fault that is distinguishable from correct behavior and another future trajectory that is indistinguishable from correct behavior. Depending on the severity of the fault, of the estimated "probability" of the distinguishable future trajectory and of the impact of the fault in the indistinguishable future trajectory, he can thus decide to change or add sensors and check again both manifestability and diagnosability.

6 - Conclusion

6.1 . Thesis Overview

In this work, we explored how to design a safe system more economically. To achieve it, we figured out different solutions from different perspectives. Firstly, in order to improve the efficiency of our previous SMT-based approach used to verify time bounded diagnosability on timed automata, we tried to take advantage of the RECAR framework by making use of approximations. Secondly, we focused on how to design a diagnosable system with delay blocks. Thirdly, we proposed a weaker system property than diagnosability to ensure that a fault in the system will manifest itself at some point in the future. Our work is structured around the two following system properties.

6.1.1 . Diagnosability

First of all, we analyzed the diagnosability of real-time systems based on SMT. The diagnosability problem of discrete event systems has received considerable attention in the literature. However, up to now litter work takes into account explicit time constraints during this analysis, which are however naturally present in the real-time systems and thus cannot be neglected considering their impact on this property. Thus, inspired by the way checking diagnosability of finite state automata for discrete event systems using SAT technology, we analyzed the diagnoability of timed automata for real-time systems using SMT technology, which can provide a natural symbolic representation for timed automata. To do this, both TA and the sufficient and necessary condition of this property, i.e., the existence of a timed critical pair, are encoded in SMT as a logic formula interpreted in linear arithmetic theory whose satisfiability witnesses bounded non-diagnosability. We have not only theoretically proved the correctness of our approach but also demonstrated its feasibility by applying it on different versions of two benchmarks selected from literature.

Diagnosability verification for RTSs with RECAR-like approach

Since the diagnosability verification problem for timed automata is known to be PSPACE-complete and RECAR appears promising in dealing with PSPACE-complete problems, in order to improve the efficiency of diagnosability verification for RTS, we proposed a RECAR-like approach for diagnosability verification of timed automata by taking the advantage of the RECAR framework. In this part of the study, first, to realize CEGAR-over and CEGAR-under, we defined different parameterized over- and under-approximations of the original problem. Then, we

defined a switching function to construct a RECAR-like approach, i.e., to alternate between the two kinds of CEGAR. The RECAR-like approach precisely interleaves both kinds of approximations: each one is performed with the information retrieved from solving the previous one. Finally, to show the feasibility and efficiency of our approach, we performed several experiments on different benchmarks. Based on the experimental results, we concluded that the key factor in the complexity of our method is the length (i.e., the value of the bound in this bounded model checking framework) of the critical pair that is looked for, which also provides us with directions for future research.

Designing diagnosable DESs

One aim of diagnosabiliy verification is to design a diagnosable system, however, it is not easy to ensure in practice that a designed system is diagnosable. In some cases, if the system has revealed as non-diagnosable, one classical way to solve this problem is to reconfigure the observable events, but again is in general too expensive in terms of sensors and may require a lot of iterations. So we proposed a new non-intrusive way to make a non-diagnosabile system diagnosable by merely adding delay blocks on some observable events, keeping the original system structure. In order to efficient eliminate all existing critical pairs by adding delay blocks as less as possible and without generating new ones, we calculated the minimum number of transitions of the normal diagnoser, where delay blocks will be added, according to the max-flow min-cut theorem. This approach is encoded into an SMT formula, whose correctness and efficiency are demonstrated by our experimental results.

6.1.2 . Manifestability

Since diagnosability generally requires a high number of sensors, it is often too expensive to develop a diagnosable system. In order to bring an alternative to diagnosability analysis, whose satisfaction is very demanding in terms of sensors placement, we have defined manifestabiliy property, that represents the weakest requirement on observations for having a chance to identify online fault occurrences and can be verified at design stage. In our study, we not only defined manifestability, but also addressed its formal verification for both DESs and real-time systems modeled as TA. For this, we have constructed different structures from the system model.

Manifestability for DESs

For discrete event systems (modeled as finite state automata), first, we have defined (strong) manifestability, then we proposed an algorithm with PSPACE

complexity to automatically verify it. Furthermore, we have demonstrated that manifestability checking boils down to languages inclusion checking and that the manfestability problem is PSPACE-complete for finite automata, for which we have provided preliminary experimental results showing the efficiency and scalability of this approach.

Manifestability for RTSs

For real-time systems (modeled as timed autaomata), we extended our manfestability verification approach for DESs to RTSs. To do this, we redefined (strong) manifestability by taking into account time constraints, then we proved that manifestability checking problem is undecidable for TA. Thus we further defined SS-DTA, a subclass of deterministic TA, for which this problem becomes PSPACE-complete. It is thus encoded into an SMT formula, and checked automatically by SMT solver Z3, finally, the preliminary experimental results showed the efficiency and scalability of this approach.

We are at the end of this thesis and we can sum-up quickly our contributions as: a new system property called manifesability that describes the capability of a system to manifest a fault occurrence in at least one of its future behavior; a new method to designing diagnosable discrete event systems using delay blocks; a new RECAR-like framework to optimize diagnosability verification for real-time systems.

Obviously, this thesis raises more questions than it solves and we give in the next sections few directions that we consider worth exploring.

6.2 . Future Work

We see several important research lines to explore in order to enhance our results.

- In order to optimize RECAR-like framework for diagnosability verification of TA, try to play with time constraints expressed in the guards and invariants, i.e., for over-approximations, tightening these constraints and, for under-approximations, relaxing these constraints.
- Investigate the conditions under which the (length) bounded non-diagnosability implies also non-diagnosability in RTSs, i.e., have an estimate of a sufficient length upper bound.
- Find out a larger subclass of TA than SS-DTA, for which manifesatbility problem is decidable and relate this problem to opacity.
- Extend our diagnosability checking framework to distributed systems by considering communicating automata.

- Investigate how to design a diagnosable system with general delay blocks. In our current work, we assumed that observable events all occur at the same time when they are not deferred and all at a same later time when they are deferred, considering thus a single delay block with only one delay. Even if it can theoretically solve most problems for DESs, is not realistic, as in practice, event occurrence is never instantaneous. In order to handle such various occurrence times and generalize this method to RTSs, investigate how to extend the current approach to general delay blocks, i.e., delay blocks with various delays. This requires computing the delays in order to avoid as far as possible creating new critical pairs.

- Study manifestability problem for distributed systems with a modular method, more interestingly, by taking into account probabilistic aspects.

- Since the complexity of manifestability verification problem is PSPACE-complete for finite state automata and for a deterministic subclass of timed automata, try to analyse manifestability with RECAR-like approach for both kinds of systems.

- Study other related properties, such as predictability, when taking into account time constraints.

A - French synthesis

Le diagnostic de pannes est une tâche cruciale et difficile dans le contrôle automatique des systèmes complexes, dont l'efficacité dépend d'une propriété du système appelée diagnosticabilité. La diagnosticabilité décrit la propriété du système permettant de déterminer dès la phase de conception si un défaut donné se produisant en ligne sera identifiable avec certitude sur la base des observations disponibles, ce qui est une alternative aux tests qui ne peuvent que montrer la présence de défaillances sans garantir leur absence. Le problème de la diagnosticabilité des systèmes à événements discrets a reçu une attention considérable dans la littérature, mais peu nombreux sont les travaux qui prennent en compte des contraintes de temps explicites lors de cette analyse. Or de telles contraintes sont naturellement présentes dans les systèmes réels et ne peuvent être négligées compte tenu de leur impact sur cette propriété. Nous avions proposé dans notre travail de master une nouvelle approche à base de SMT (Satisfiability Modulo Theories) pour vérifier la diagnosticabilité en temps borné sur les automates temporisés. L'idée était d'encoder en SMT une condition nécessaire et suffisante de diagnosticabilité dans une démarche de model checking borné. Dans ce travail, nous n'avions pas seulement prouvé la correction de notre approche mais aussi amélioré son efficacité grâce à une indexation des transitions du système réduisant la taille de l'espace de recherche défini par la borne et montré sa faisabilité en l'appliquant à des benchmarks. Néanmoins, le problème étant PSPACE-complet, nous nous heurtions au passage à l'échelle lorsque le nombre d'états du système s'accroît. Suivant le schéma de l'algorithme RECAR (exploration et vérification récursives de raffinements d'abstraction), qui apparaît prometteur pour traiter des problèmes au delà de NP, en particulier des problèmes PSPACE-complets, nous proposons à présent un algorithme de type RECAR incluant une extension incrémentale de ce travail fondée sur l'utilisation de sur- et sous-approximations paramétrées généralisant la méthode CEGAR (raffinement d'abstraction guidé par un contre-exemple). Afin d'utiliser CEGAR pour une terminaison précoce de la boucle d'itération des raffinements des sur- (resp., sous-) approximations lorsque la formule originale est satisfaisable (resp., insatisfaisable), nous définissons trois types de sur-approximations (resp., deux types de sous-approximations). Ensuite nous alternons CEGAR avec sur-approximations et CEGAR avec sous-approximations en définissant une fonction de commutation pour décider du passage d'un type d'approximations à l'autre. Nous montrons l'amélioration apportée au travers de résultats expérimentaux et identifions la longueur de la paire critique recherchée comme le facteur clé de la complexité.

Néanmoins, la diagnosticabilité est une propriété assez forte, qui nécessite généralement un nombre élevé de capteurs. Par conséquent, il n'est pas rare que

le développement d'un système diagnosticable soit trop coûteux. Afin de garantir dès la conception un certain niveau de sûreté de fonctionnement de manière économique et efficace, nous proposons deux approches.

La première consiste à concevoir des systèmes à événements discrets diagnosticables en utilisant des blocs de retard. La diagnosticabilité étant une propriété critique d'un système, il est en effet important que la phase de conception aboutisse à un système diagnosticable, ce qui s'avèrera déterminant pour le diagnostic ultérieur du système. Mais cette tâche est loin d'être triviale et jusqu'à présent la plupart des travaux se sont focalisés sur la vérification de la diagnosticabilité, sans répondre à la question : que faire si un système se révèle comme non diagnosticable ? On doit en fait dans ce cas étudier comment le rendre diagnosticable. Une manière classique est d'ajouter des capteurs, ce qui est en général coûteux et peut requérir beaucoup d'itérations. Un autre moyen repose sur des événements contrôlables qui peuvent contraindre les comportements du système de façon à ce que les comportements autorisés sont diagnosticables. Nous proposons une nouvelle manière non intrusive de rendre diagnosticable un système non diagnosticable en ajoutant simplement des blocs de retard sur certains événements observables, différant ainsi leurs observations. Pour autant que nous le sachions, il s'agit de la première tentative de suppression de la non diagnosticabilité avec des blocs de retard sans utiliser d'événements contrôlables et sans modifier la structure des systèmes. Pour des raisons de simplicité, ce travail a été conduit sur des systèmes à événements discrets classiques sans contraintes temporelles, donc seul l'ordre des événements observables est pris en compte. On calcule, à l'aide du théorème flot-max coupe-min, un ensemble minimal d'événements observables dont le retard rend diagnosticable un système non diagnosticable. Plus précisément, on adapte le concept d'automates finis avec blocs de retard pour éliminer toutes les paires de trajectoires témoignant de la non diagnosticabilité en prenant soin de ne pas en créer de nouvelles, de façon à ce que toute trajectoire avec présence d'un défaut puisse être distinguée par les observations (avec prise en compte des retards) de toutes les trajectoires normales. Notre approche est codée dans une formule SMT, dont l'exactitude et l'efficacité sont démontrées par nos résultats expérimentaux.

La seconde consiste à analyser une nouvelle propriété d'un système appelée manifestabilité, moins forte que la diagnosticabilité, afin de réaliser un compromis entre le coût, c'est-à-dire un nombre raisonnable de capteurs, et la possibilité d'observer la manifestation d'un défaut. La manifestabilité est une exigence plus faible sur les observations du système pour avoir une chance d'identifier l'occurrence des défauts en ligne et peut être vérifiée au stade de la conception. Intuitivement, cette propriété garantit qu'un système défectueux ne peut pas toujours apparaître sain, c'est-à-dire qu'il a au moins un comportement futur après l'apparition d'un défaut (et non pas tous les comportements futurs comme pour la diagnosticabilité)

qui se distingue par l'observation de tous les comportements normaux. Naturellement on doit continuer à se reposer sur la diagnosticabilité pour les exigences de sûreté en ligne, c'est-à-dire pour les défauts qui peuvent avoir des conséquences dramatiques s'ils ne sont pas détectés à temps pour déclencher des actions correctrices. Mais pour tous les autres défauts qui ne nécessitent pas d'être détectés dès leur apparition (c'est-à-dire dont la conséquence est un fonctionnement dégradé mais acceptable du système qui nécessitera des actions de maintenance dans un certain futur proche), la vérification de leur manifestabilité, moins coûteuse en termes de capteurs nécessaires, est suffisante sous l'hypothèse probabiliste qu'aucun comportement décrit du système n'a une probabilité nulle. Nous définissons d'abord la manifestabilité des systèmes à événements discrets modélisés par des automates à états finis, proposons un algorithme de complexité PSPACE pour la vérifier automatiquement et prouvons que le problème de vérification de la manifestabilité lui-même est PSPACE- complet. Les résultats expérimentaux montrent la faisabilité de notre algorithme d'un point de vue pratique. Ensuite, nous définissons la manifestabilité des systèmes temps-réel modélisés par des automates temporisés en tenant compte des contraintes de temps, et étendons notre approche pour vérifier la manifestabilité de ces systèmes, prouvant qu'elle est indécidable en général mais, sous certaines conditions restreintes, devient PSPACE- complet. Enfin, nous encodons cette propriété dans une formule SMT, dont la satisfaisabilité témoigne de la manifestabilité, avant de présenter des résultats expérimentaux montrant le passage à l'échelle de notre approche.

B - Publications

[1] P. Dague, L. He, and L. Ye. How to be sure a faulty system does not always appear healthy? *Innovations in Systems and Software Engineering*, 16(2):121–142, 2020.

[2] L. Ye, P. Dague, and L. He. Manifestability verification of discrete event systems. In *Proceedings of the 30th International Workshop on Principles of Diagnosis DX'19*, 2019.

[3] L. He, L. Ye, and P. Dague. SMT-based diagnosability analysis of real-time systems. In *Proceedings of the 10th Symposium on Fault Detection, Supervision and Safety for Technical Processes, IFAC SAFEPROCESS 2018*, 2018.

C - Software

In chapter 2, we showed how to encode the verification of the bounded diagnosability (where a a counterexample is defined as a critical pair, please see the paper DiagnosabilityDES.pdf) for finite automata in smt with the files available in https://github.com/lu-1993/Diagnosability, and a very simple example stored in model.txt. This simple system contains six events, where three observable events are represented by integers 1-3 and two unobservable normal events by 4-5 and one unobservable faulty event by 6. There are ten transitions in this system that one can find in the file model.txt beginning from line 2 (one transition per line). The file $diagWithQuantifier.smt$ contains the smt formula simplified from that is presented in the paper DiagnosabilitySMTtimedAutomata.pdf, the latter is for timed automata. The file $diagNoQuantifier.smt$ is a quantifier-free version for the same example. And result.txt contains a model satisfying the smt formula, i.e., a critical pair which is a pair of trajectories of the system where only one of them contains the fault while both of them have the same observations.

An optimized version is also available in the OptimizedDiagSMT package, a python package that generates automatically an smt formula in an optimized way in terms of the path length for this problem with different parameters for a given system. To obtain the verification result, it is enough to run the python file $main.py$ that generates the smt formula, which is checked by calling z3 in an optimized way, whose results are then analyzed before returning the final results.

For the example in model.txt, the parameters are: Bound 5 K_value 2 observable=o1,o2,o3 unobservable=un1,un2 fault=f

The final result is:

If (7 o3 8) in model.txt, return:

SAT

critical pair: the length of faulty path is: 4

[' 1', 'o1', ' 2', 'o2', ' 3', 'f', ' 4', 'un1', ' 5', 'o3', ' 5']

[' 1', 'o1', ' 2', 'o2', ' 6', 'un2', ' 7', 'o3', ' 8']

observations: [" 'o1'", " 'o2'", " 'o3'"]

If (7 o2 8) in model.txt, return:

UNSAT

possible critical pair : the length of faulty path is: 4

[' 1', 'o1', ' 3', 'f', ' 4', 'un1', ' 5', 'o3', ' 5']

[' 1', 'o1', ' 2']

blocked in: 'o3'

possible critical pair : the length of faulty path is: 5

[' 1', 'o1', ' 2', 'o2', ' 3', 'f', ' 4', 'un1', ' 5', 'o3', ' 5']

[' 1', 'o1', ' 2', 'o2', ' 6', 'un2', ' 7', 'o2', ' 8', 'o2', ' 8']

blocked in: 'o3'

In chapter 3, we showed the feasibility of the RECAR-like algorithm and we pushed our code in https://github.com/lu-1993/diaForTA.

- $cegarWithoutIncT.py$ is checking diagnosability of timed automata without using CEGAR algorithm.
- $cegarwithBTnew.py$ is CEGAR-over algorithm with parameter bound changeable.
- $cegarwithTTnew.py$ is CEGAR-over algorithm with parameter transition set changeable.
- $cegarwithOT\_over.py$ is CEGAR-over algorithm with parameter observable event set changeable.
- $cegarwithDTnew.py$ is CEGAR-under algorithm with parameter time after first fault occurrence changeable.
- $cegarwithOT\_under.py$ is CEGAR-under algorithm with observable event set changeable.
- $semi\_under.py$ is semi-equivalence program.
- $switch\_new.py$ is RECAR-like algorithm.

In the console enter the command line $python3\ name.py\ model.txt$ that can run different programs, where $name.py$ is the program name and $model.txt$ is a txt file that stores different models.

In chapter 4, to show the feasibility of our approach, we carried out a prototype implementation done in Python by using the SMT solver Z3. All our experimental results are obtained by running our programs on a Mac OS laptop with the processor 2.7 GHz Intel Core i5, 8 Go 1600 MHz DDR3 of memory. Source code and experiments are available at https://github.com/lu-1993/Designing-DIA-via-delay-blocks.

Run python program through console: $python3\ diaViaDB.py\ model.txt$.

In chapter 5, to show the feasibility of our approach to verify manifestability, we pushed our programs at https://github.com/lu-1993/manifestability.

- $modeleg1.txt$ stores the fault(refined) diagnoser.
- $modeleg2.txt$ stores the fault pair verifier.
- $testmani.py$ is the main program for checking manifestability of a SS-DTA.

To run our program, enter the command line: $python3\ testmani.py\ model.txt$.

Bibliography

[1] A. Agarwal, A. Madalinski, and S. Haar. Effective Verification of Weak Diagnosability. In *Proceedings of the 8th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS'12)*, pages 636–641. IFAC, 2012.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, USA, 1993.

[3] R. Alur and D. Dill. The theory of timed automata. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 45–73. Springer, 1991.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[5] R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems*, pages 1–24. Springer, 2004.

[6] A. Armando and E. Giunchiglia. Embedding complex decision procedures inside an interactive theorem prover. *Annals of Mathematics and Artificial Intelligence*, 8(3):475–502, 1993.

[7] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence IJCAI'09*, pages 399–404, 2009.

[8] B. Badban and M. Lange. Exact incremental analysis of timed automata with an smt-solver. In *Proceedings of International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'11)*, volume 6919 of *Lecture Notes in Computer Science*. Springer, 2011.

[9] E. Badouel, M. Bednarczyk, A. Borzyszkowski, B. Caillaud, and P. Darondeau. Concurrent secrets. *Discrete Event Dynamic Systems*, 17(4):425–446, 2007.

[10] C. Baier, N. Bertrand, P. Bouyer, and T. Brihaye. When are timed automata determinizable? In *Proceedings of the 36th International Colloquium on Automata, Languages, and Programming ICALP'09*, pages 43–54. Springer, 2009.

[11] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[12] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Handbook of satisfiability, Biere, Heule and van Maaren, eds.*, chapter 12, Satisfiability modulo theories, pages 737–797. IOS press, 2008.

[13] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[14] C. Barrett, M. Deters, L. De Moura, A. Oliveras, and A. Stump. 6 years of smt-comp. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[15] M. Basarkar, X. Pang, L. Wang, P. Haves, and T. Hong. Modeling and simulation of HVAC faults in EnergyPlus. In *Proceedings of Building Simulation 2011: 12th Conference of International Building Performance Simulation Association*, pages 2897–2903, Jan 2011.

[16] J. C. Basilio, S. T. S. Lima, S. Lafortune, and M. V. Moreira. Computation of minimal event bases that ensure diagnosability. *Discrete Event Dynamic Systems: Theory and Applications*, 22(3):249–292, 2012.

[17] B. Bérard, P. Gastin, and A. Petit. On the power of non-observable actions in timed automata. In *Proceedings of 13th Annual Symposium on Theoretical Aspects of Computer Science STACS'96*, pages 257–268. Springer, 1996.

[18] N. Bertrand, E. Fabre, S. Haar, S. Haddad, and L. Hélouët. Active diagnosis for probabilistic systems. In *International Conference on Foundations of Software Science and Computation Structures*, pages 29–42. Springer, 2014.

[19] N. Bertrand, S. Haddad, and E. Lefaucheux. Foundation of diagnosis and predictability in probabilistic systems. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, pages 417–429, 2014.

[20] N. Bertrand, S. Haddad, and E. Lefaucheux. Diagnosis in infinite-state probabilistic systems. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 37:1–37:15, 2016.

[21] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320, 1999.

[22] B. Bittner, M. Bozzano, and A. Cimatti. Automated synthesis of timed failure propagation graphs. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 972–978, 2016.

[23] B. Bittner, M. Bozzano, A. Cimatti, and G. Zampedri. Automated verification and tightening of failure propagation models. In *Proceedings of the 30th Conference on Artificial Intelligence (AAAI'16)*, pages 907–913, 2016.

[24] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Proceedings of the International Conference on Computer Aided Verification*, pages 454–464. Springer, 2001.

[25] F. Bonchi and D. Pous. Checking NFA Equivalence with Bisimulations Up to Congruence. In *Proceedings of 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-2013)*, pages 457–468. ACM, 2013.

[26] P. Bouyer, F. Chevalier, and D. D'Souza. Fault diagnosis using timed automata. In *Proceedings of International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'05)*, Lecture Notes in Computer Science. Springer, 2005.

[27] M. Bozzano, A. Cimatti, M. Gario, and A. Micheli. Smt-based validation of timed failure propagation graphs. In *Proceedings of the 29th Conference on Artificial Intelligence (AAAI'15)*, pages 3724 –3730, 2015.

[28] L. Brandán Briones, A. Lazovik, and P. Dague. Optimizing the system observability level for diagnosability. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation ISoLA'08*, 2008.

[29] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[30] F. Cassez. The dark side of timed opacity. In *Proceedings of the 3rd International Conference on Information Security and Assurance ISA'09*, pages 21–30, 2009.

[31] F. Cassez, J. Dubreil, and H. Marchand. Dynamic observers for the synthesis of opaque systems. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis ATVA'09*, pages 352–367, 2009.

[32] M. Chankate, A. Philippot, V. Carre-Menetrier, and P. Marangé. Checking diagnosability on centralized model of the system. In *Proceedings of the 3rd IEEE International Conference on Control, Automation and Diagnosis ICCAD'19*, pages 386–391, 2019.

[33] E. Chanthery and Y. Pencolé. Monitoring and active diagnosis for discrete-event systems. In *Proceedings of the 7th IFAC International Symposium on Fault Detection, Supervision and Safety for Technical Processes SafeProcess'09*, 2009.

[34] K. Chatterjee, T.A., Henzinger, and V.S.Prabhu. Finite automata with time-delay blocks. *Proceedings of the 10th ACM international conference on Embedded software EMSOFT'12*, pages 43–52, 2012.

[35] A. Cimatti, C. Pecheur, and R. Cavada. Formal Verification of Diagnosability via Symbolic Model Checking. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence IJCAI'03*, pages 363–369, 2003.

[36] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification CAV 2000*, pages 154–169. Lecture Notes in Computer Science, vol. 1855, Springer, 2000.

[37] L. Console, C. Picardi, and M. Ribaudo. Process algebras for systems diagnosis. *Artificial Intelligence*, 142(1):19–51, 2002.

[38] P. Dague, L. He, and L. Ye. How to be sure a faulty system does not always appear healthy? Fault manifestability analysis for discrete event and timed systems. *Innovations in Systems and Software Engineering (ISSE), a NASA journal*, 2019.

[39] A. Darwiche and G. Provan. Exploiting system structure in model-based diagnosis of discrete-event systems. In *Proceedings of the 7th International Workshop on Principles of Diagnosis*, 1996.

[40] J. Davies and F. Bacchus. Exploiting the power of mip solvers in maxsat. In *International conference on theory and applications of satisfiability testing*, pages 166–181. Springer, 2013.

[41] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[42] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[43] R. Debouk, R. Malik, and B. Brandin. A modular architecture for diagnosis of discrete event systems. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 1, pages 417–422. IEEE, 2002.

[44] V. Diekert, P. Gastin, and A. Petit. Removing ϵ-transitions in timed automata. In *Proceedings of 14th Annual Symposium on Theoretical Aspects of Computer Science STACS'97*, pages 583–594, 1997.

[45] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of SAT'03*, pages 502–518, 2003.

[46] Y. Falcone and H. Marchand. Enforcement and validation (at runtime) of various notions of opacity. *Discrete Event Dynamic Systems*, 25(4):531–570, 2014.

[47] O. Finkel. Undecidable problems about timed automata. In *Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems FORMATS 2006*, pages 187–199. Springer, 2006.

[48] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[49] G. Gardey, J. Mullins, and O. Roux. Non-interference control synthesis for security timed automata. *Electronic Notes in Theoretical Computer Science*, 180(1):35–53, June 2007.

[50] V. Germanos, S. Haar, V. Khomenko, and S. Schwoon. Diagnosability under weak fairness. In *Proceedings of the 14th International Conference on Application of Concurrency to System Design ACSD'14*. IEEE Computer Society Press, 2014.

[51] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures—the case study of modal k. In *Proceedings of the International Conference on Automated Deduction*, pages 583–597. Springer, 1996.

[52] A. Grastien, A. Anbulagan, J. Rintanen, and E. Kelareva. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proceedings of the 22nd National Conference on Artificial Intelligence AAAI'07*, pages 305–310, 2007.

[53] S. Haar, S. Haddad, T. Melliti, and S. Schwoon. Optimal constructions for active diagnosis. In A. Seth and N. Vishnoi, editors, *Proceedings of the 33rd Conference on Foundations of Software Technology and Theoretical Computer Science FSTTCS'13*, volume 24, pages 527–539, 2013.

[54] S. Haar, S. Haddad, T. Melliti, and S. Schwoon. Optimal construction for active diagnosis. *J Comput Syst Sci.*, 83(1):101–120, 2017.

[55] L. He, L. Ye, and P. Dague. SMT-based diagnosability analysis of real-time systems. In *Proceedings of the 10th Symposium on Fault Detection, Supervision and Safety for Technical Processes, IFAC SAFEPROCESS 2018*, 2018.

[56] F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Lazy abstractions for timed automata. In *Proceedings of the 25th International Conference on Computer Aided Verification CAV 2013*, pages 990–1005, 2013.

[57] H. Ibrahim. *Analyse à base de SAT de la diagnosticabilité et de la prédictabilité des systèmes à événements discrets centralisés et distribués*. PhD thesis, Université Paris-Saclay, 2016.

[58] R. Jacob, J.-J. Lesage, and J.-M. Faure. Opacity of discrete event systems: models, validation and quantification. In *Proceedings of the 5th IFAC Workshop on Dependable Control of Discrete Systems DCDS'15*, pages 174–181, 2015.

[59] S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial algorithm for testing diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 46(8):1318–1321, 2001.

[60] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence ECAI'92*, pages 359–363, 1992.

[61] R. Kindermann, T. Junttila, and I. Niemela. Beyond lassos: Complete smt-based bounded model checking for timed automata. In *Proceedings of Joint FMOODS 2012 and FORTE 2012*, volume 7273 of *Lecture Notes in Computer Science*. Springer, 2012.

[62] J. Kleinberg and E. Tardos. Network flow. In *Algorithm design*, pages 337–411. Pearson Education India, 2006.

[63] J. Lagniez, D. Le Berre, T. de Lima, and V. Montmirail. A recursive shortcut for CEGAR: application to the modal logic K satisfiability problem. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence IJCAI'17*, pages 674–680, 2017.

[64] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[65] Y. Li and Z. O'Neill. A critical review of fault modeling of HVAC systems in buildings. *Building Simulation*, 11(5):953–975, 2018.

[66] F. Lin. Opacity of discrete event systems and its applications. *Automatica*, 47(3):496–503, 2011.

[67] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[68] L. d. Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[69] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[70] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[71] D. Papineau. *Philosophical naturalism*. Blackwell, Oxford, 1993.

[72] C. Pecheur, A. Cimatti, and R. Cimatti. Formal verification of diagnosability via symbolic model checking. In *Proceedings of the Workshop on Model Checking and Artificial Intelligence (MoChArt-2002), Lyon, France*, 2002.

[73] Y. Pencolé. Diagnosability analysis of distributed discrete event systems. In *Proceedings of the 16th European conference on artificial intelligence ECAI'04*, pages 43–47, 2004.

[74] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[75] J. Rintanen. Diagnosers and Diagnosability of Succinct Transition Systems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence IJCAI'07*, pages 538–544, 2007.

[76] J. Rintanen and A. Grastien. Diagnosability testing with satisfiability algorithms. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence IJCAI'07*, pages 532–537, 2007.

[77] A. Saboori. *Verification and enforcement of state-based notions of opacity in discrete event systems*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 2011.

[78] A. Saboori and C.-N. Hadjicostis. Notions of security and opacity in discrete event systems. In *Proceedings of the 46th IEEE Conference on Decision and Control CDC'07*, pages 5056–5061, 2007.

[79] A. Saboori and C.-N. Hadjicostis. Verification of infinite-step opacity and analysis of its complexity. In *Proceedings of the 2nd IFAC Workshop on Dependable Control of Discrete Systems DCDS'09*, pages 46–51, 2009.

[80] A. Saboori and C.-N. Hadjicostis. Verification of infinite-step opacity and complexity considerations. *IEEE Transactions on Automatic Control*, 57(5): 1265–1269, 2012.

[81] A. Saboori and C.-N. Hadjicostis. Verification of initial-state opacity in security applications of discrete event systems. *Information Sciences*, 246: 115–132, 2013.

[82] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on automatic control*, 40(9):1555–1575, 1995.

[83] M. Sampath, S. Lafortune, and D. Teneketzis. Active diagnosis of discrete-event systems. *IEEE Trans. Autom. Control.*, 43(7):908–929, 1998.

[84] A. Schumann and Y. Pencolé. Scalable diagnosability checking of event-driven system. In *Proceedings of the 20th international joint conference on artificial intelligence IJCAI'07*, pages 575–580, 2007.

[85] A. Schumann, J. Huang, et al. A scalable jointree algorithm for diagnosability. In *Proceedings of the 23rd American national conference on artificial intelligence AAAI'08*, pages 535–540, 2008.

[86] S. Shu and F. Lin. Detectability of Discrete Event Systems with Dynamic Event Observation. *Systems and Control Letters*, 59(1):9–17, 2010.

[87] S. Shu and F. Lin. I-Detectability of Discrete-Event Systems. *IEEE T. Automation Science and Engineering*, 10(1):187–196, 2013.

[88] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *The Best of ICCAD*, pages 73–89. Springer, 2003.

[89] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2-3):217–237, 1987.

[90] N. Sorensson and N. Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

[91] P. Struss. Fundamentals of model-based diagnosis of dynamic systems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence IJCAI'97*, pages 480–485, 1997.

[92] X. Su, M. Zanella, and A. Grastien. Diagnosability of discrete-event systems with uncertain observations. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence IJCAI'16*, pages 1265–1271, 2016.

[93] D. Thorsley and D. Teneketzis. Diagnosability of stochastic discrete-event systems. *IEEE Trans. Automat. Contr.*, 50(4):476–492, 2005.

[94] S. Tripakis. Fault diagnosis for timed automata. In *Proceedings of the International symposium on formal techniques in real-time and fault-tolerant systems*, pages 205–221. Springer, 2002.

[95] G. S. Tseitin. The upper bounds of enumerable sets of constructive real numbers. *Trudy Matematicheskogo Instituta imeni VA Steklova*, 113:102–172, 1970.

[96] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[97] J. Wang, M. Agrawala, and M. F. Cohen. Soft scissors: an interactive tool for realtime high quality matting. *ACM SIGGRAPH 2007 papers*, 26(3): 9–14, 2007.

[98] L. Wang, N. Zhan, and J. An. The opacity of real-time automata. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2845–2856, 2018.

[99] Y.-C. Wu and S. Lafortune. Comparative analysis of related notions of opacity in centralized and coordinated architectures. *Discrete Event Dynamic Systems*, 23(3):307–339, 2013.

[100] L. Ye and P. Dague. Diagnosability analysis of discrete event systems with autonomous components. In *Proceedings of the 19th European conference on Artificial Intelligence ECAI'10*, pages 105–110, 2010.

[101] L. Ye, P. Dague, D. Longuet, L. B. Briones, and A. Madalinski. How to be sure a faulty system does not always appear healthy? In *International Conference on Verification and Evaluation of Computer and Communication Systems*, pages 114–129. Springer, 2018.

[102] X. Yin and S. Lafortune. A new approach for the verification of infinite-step and k-step opacity using two-way observers. *Automatica*, 80:162–171, 2017.

[103] T.-S. Yoo and S. Lafortune. NP-completeness of sensor selection problems arising in partially observed discrete-event systems. *IEEE Trans. Autom. Control.*, 47(9):1495–1499, 2002.

[104] A. Zbrzezny and A. Półrola. Sat-based reachability checking for timed automata with discrete data. *Fundamenta Informaticae*, 79(3-4):579–593, 2007.

[105] B. Zhang, S. Shu, and F. Lin. Polynomial algorithms to check opacity in discrete event system. In *Proceedings of the 24th Chinese Control and Decision Conference CCDC'12*, pages 763–769, 2012.