CHALMERS

INCREMENTAL FAULT DIAGNOSABILITY AND SECURITY/PRIVACY VERIFICATION

MONA NOORI-HOSSEINI

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2020

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Incremental Fault Diagnosability and Security/Privacy Verification

MONA NOORI-HOSSEINI

Systems and Control Group
Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2020

Incremental Fault Diagnosability and Security/Privacy Verification
MONA NOORI-HOSSEINI
ISBN: 978-91-7905-287-4

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4575
ISSN 0346-718X

Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Telephone: +46 (0)31 772 1000

This thesis has been prepared using LaTeX.

Printed by Chalmers Reproservice
Göteborg, Sweden, April 2020

To my beloveds Behrooz,
Nickaan & Edwin

Abstract

Dynamical systems can be classified into two groups. One group is continuous-time systems that describe the physical system behavior, and therefore are typically modeled by differential equations. The other group is discrete event systems (DES)s that represent the sequential and logical behavior of a system. DESs are therefore modeled by discrete state/event models.

DESs are widely used for formal verification and enforcement of desired behaviors in embedded systems. Such systems are naturally prone to faults, and the knowledge about each single fault is crucial from safety and economical point of view. Fault diagnosability verification, which is the ability to deduce about the occurrence of all failures, is one of the problems that is investigated in this thesis. Another verification problem that is addressed in this thesis is security/privacy. The two notions current-state opacity and current-state anonymity that lie within this category, have attracted great attention in recent years, due to the progress of communication networks and mobile devices.

Usually, DESs are modular and consist of interacting subsystems. The interaction is achieved by means of synchronous composition of these components. This synchronization results in large monolithic models of the total DES. Also, the complex computations, related to each specific verification problem, add even more computational complexity, resulting in the well-known state-space explosion problem.

To circumvent the state-space explosion problem, one efficient approach is to exploit the modular structure of systems and apply incremental abstraction. In this thesis, a unified abstraction method that preserves temporal logic properties and possible silent loops is presented. The abstraction method is incrementally applied on the local subsystems, and it is proved that this abstraction preserves the main characteristics of the system that needs to be verified.

The existence of shared unobservable events means that ordinary incremental abstraction does not work for security/privacy verification of modular DESs. To solve this problem, a combined incremental abstraction and observer generation is proposed and analyzed. Evaluations show the great impact of the proposed incremental abstraction on diagnosability and security/privacy verification, as well as verification of generic safety and liveness properties. Thus, this incremental strategy makes formal verification of large complex systems feasible.

Keywords: Incremental abstraction, Formal verification, Temporal logic, Fault diagnosability, Opacity, Anonymity, Automation, Discrete event systems.

Acknowledgments

I feel extremely fortunate to have met so many wonderful, inspiring and kind people on this PhD journey, whom I would like to express my sincerest thanks to.

First and foremost, I would like to express my most humble gratitude to my supervisor, Professor Bengt Lennartson, for giving me the opportunity to work in his research group. Your patience, motivation, and guidance helped me a lot. I appreciate the life-lessons I learned from you, to simplify things which is when the miracle happens, and also to enjoy what I am doing.

Thanks to all the seniors in the Automation group, Professors Martin Fabian, Knut Åkesson and Petter Falkman, for all interesting technical and non-technical talks. Martin, thanks for being so friendly and supportive. I also would like to thank the former and current PhD students in the group and my friends in the Electrical Engineering department for the fun time and laughs we had together and the great memories we have made.

I also would like to thank the co-authors of our papers, specially Maria Paola Cabasino, Carla Seatzu and Christoforos N. Hadjicostis for sharing such valuable knowledge and experiences. Furthermore, I would like to thank Xudong Liang, for his great work during his master thesis in demonstrating the performance of our methods. I would like to express my appreciation to the administration team, Mrs. Agneta Kinnander, Mrs. Natasha Adler and Mrs. Madeleine Persson, for being so helpful.

I would like to dedicate a special thanks to Martin Magnusson, my manager at the Autonomous Drive Functions team at Volvo Cars, the most caring and understanding person every employee would like to work with. My gratitude also goes to my friends and colleagues at Volvo Cars for their friendship and the great environment they have created to make the not easy period of learning new things at work, writing the thesis at home, while raising two small kids, to become easier. It has been delightful working with you.

In all different stages of your life, you need those friends who would help you forget about all the pressure, and re-energize, I have been blessed with many and with all my heart I am grateful to them. Thanks for all the good moments we had together.

My sincerest gratitude and love go to my family for all their support. Dad, thanks for teaching me to be ambitious. I really missed the moments we spent together, specially the breakfasts you prepared for us on weekends. Mom, thanks for encouraging me to think outside the box. I am always enjoying our phone calls. Your beautiful voice is like a music to my ears. Golsa and Soroush, thanks for always being there for us. The door of your home is always open to us and we are sure that we are pampered there. Golshid and Mehrdad, thanks for all your help to our medical questions.

I am so grateful and blessed for the two angles, Nickaan and Edwin, that came to my life during my PhD study. You make my days bright and joyful. Playing with you is my meditation. Now, it comes to the most important person in my life. Behrooz, I am so lucky to have you in my life. You have the most beautiful soul and the most loving heart. I am proud of you, for so perfectly taking care of everything during the time I was busy with my thesis. You are the kindest, funniest and smartest husband, and dad for our kids that I could have imagined. We made it here and it was only possible together. I love you!

Mona Noori-Hosseini
Göteborg, March 2020

List of Publications

This thesis is based on the following publications:

[Paper A] **Mona Noori-Hosseini**, Bengt Lennartson, Maria Paola Cabasino and Carla Seatzu. *A survey on efficient diagnosability tests for automata and bounded Petri nets*. Proceedings of the IEEE International Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–6, 2013.

[Paper B] **Mona Noori-Hosseini** and Bengt Lennartson. *Diagnosability verification using compositional branching bisimulation*. Proceedings of the 13th Workshop on Discrete Event Systems (WODES), pp. 245–250, 2016.

[Paper C] **Mona Noori-Hosseini** and Bengt Lennartson. *Incremental Abstraction for Diagnosability Verification of Modular Systems*. Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp. 393–399, 2019.

[Paper D] **Mona Noori-Hosseini**, Bengt Lennartson and Christoforos N. Hadjicostis. *Compositional visible bisimulation abstraction applied to opacity verification*. Proceedings of the 14th Workshop on Discrete Event Systems (WODES), pp. 434–441, 2018.

[Paper E] **Mona Noori-Hosseini**, Bengt Lennartson and Christoforos N. Hadjicostis. *Incremental Observer Abstraction for Opacity/Privacy Verification and Enforcement*. Submitted for possible journal publication. An invitation to submit a revised version has been received, 2019.

Relevant work by the author, not included in the thesis

- Bengt Lennartson and **Mona Noori-Hosseini**. *Visible bisimulation equivalence – A unified abstraction for temporal logic verification*. Proceedings of the 14th Workshop on Discrete Event Systems (WODES), pp. 400–407, 2018.

- **Mona Noori-Hosseini** and Bengt Lennartson. *Diagnosability verification using compositional branching bisimulation*. Technical report, Department of Electrical Engineering, Chalmers University of Technology, 2016.

- **Mona Noori-Hosseini** and Bengt Lennartson. *Verification of diagnosability based on compositional branching bisimulation*. Proceedings of the 19th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA), 2014.

- **Mona Noori-Hosseini** and Bengt Lennartson. *Verification of diagnosability based on compositional branching bisimulation*. Technical report, Department of Electrical Engineering, Chalmers University of Technology: R013/2014.

- Bengt Lennartson, Francesco Basile, Sajed Miremadi, Zhennan Fei, **Mona Noori-Hosseini**, Martin Fabian, and Knut Åkesson. *Supervisory Control for State-Vector Transition Models—A Unified Approach*. IEEE Transactions on Automation Science and Engineering, 11(1), pp. 33–47, 2014.

- Tord Alenljung, Bengt Lennartson and **Mona Noori-Hosseini**. *Sensor graphs for discrete event modeling applied to formal verification of PLCs*. IEEE Transactions on Control Systems Technology, 20(6), pp. 1506–1521, 2012.

- Bengt Lennartson, Sajed Miremadi, Zhennan Fei, **Mona Noori-Hosseini**, Martin Fabian, and Knut Åkesson. *State-vector transition model applied to supervisory control*. Proceedings of the IEEE International Conference on Emerging Technologies & Factory Automation (ETFA), 2012.

Acronyms

BB:	Branching Bisimulation
BBSD:	BB including State labels & explicit Divergence
BRG:	Basis Reachability Graph
CSA:	Current State Anonymity
CSO:	Current State Opacity
CTL:	Computational Tree Logic
DES:	Discrete Event System
DFA:	Deterministic Finite Automata
DSV:	Divergence-Sensitive Visible bisimulation
FNC:	Future Nondeterministic Choices
MBRG:	Modified Basis Reachability Graph
PN:	Petri Net
RG:	Reachability Graph
SB:	Stuttering Bisimulation
STS:	Source and Target State
VBE:	Visible Bisimulation Equivalence

Contents

Part I

Introductory Chapters

CHAPTER 1

Introduction and Overview of the Thesis

Physical phenomena of dynamical systems are generally modeled by ordinary or partial differential equations, which include continuous variables. However, in our everyday life of the increasingly computer-dependent world, many of the quantities we deal with are discrete, and many of the processes are driven by instantaneous events, such as sliding the screen key to unlock a smart phone, or pressing a button in an automated teller machine. Systems are often event-driven, especially when digital computers are involved. Typical examples are automated manufacturing systems and communication networks. These systems should be able to adapt swiftly to changing conditions and react rapidly to unpredictable events. Above all, these systems should perform a desired behavior based on a specification and satisfy their users. The occurrences of events in these systems can be nominal, such as opening a valve, or can be unwanted, such as a sensor failure. These systems are called *discrete event systems* (DES)s [1]. A DES includes different discrete states that are changed upon the occurrence of events.

With the progress of computer-aided and large complex industrial systems, the problems regarding safety assurance are growing. The applications of these systems are seen in our daily life. Aircraft electronic systems, computer systems, and microwave ovens are common examples of the mentioned systems. All man-made

software and hardware systems are prone to faults, and fault occurrences may result in catastrophic problems. One way to evaluate whether a system performs its nominal behavior is to verify the system specification. In order to achieve this, a model of the system is required. The two main modeling formalisms of DESs are finite state automata [2] and Petri nets [3].

To be able to detect if an unobservable fault has happened in a DES, the system model must be diagnosable. This means that it is possible to detect all unobservable fault events by a diagnoser in finite time. A main topic in verification of safety properties for DESs is therefore verification of *fault diagnosability*. Diagnosability is a necessary condition that must be satisfied to be able to construct a diagnoser.

More recently, security and privacy concerns have also been raised on the information flow of large communication networks and their diverse applications in modern technologies. One of the important information flow properties related to privacy and security is called *opacity*. While diagnosability verification is about providing sufficient information to a verifier to detect faults, privacy and security require information to an outside observer or intruder to be hidden, so that secrets are not revealed.

1.1 Problem Statement

The main effort to be able to verify the aforementioned problems for larger systems is to develop algorithms with reduced computational complexity. Since systems are getting more complex, traditional test-based techniques might not be enough to ensure whether DESs perform their nominal task correctly. Formal verification techniques are capable of guaranteeing a specification for larger state spaces, and can mathematically prove whether requirement specifications are fulfilled or not. This can be relatively easily done for small and simple DESs. However, for complex DESs that are usually modular and consist of interacting subsystems, formal verification procedures often need to check millions of states or even more. With limited available time and memory, these problems lead to the *state-space explosion* problem [4].

To mitigate the well-known state-space explosion problem, one way is to develop algorithms and methods with polynomial complexity, and combine them with efficient abstraction methods. The application of the reduction methods on modular systems can be done incrementally. This *incremental abstraction* method [5], [6] is shown to be very efficient, having the important characteristic that it preserves the

main property that needs to be verified.

A more detailed background review of both diagnosability verification and opacity/anonymity verification is presented in the following paragraphs.

Fault diagnosability verification Discrete event systems are widely used, and are naturally highly prone to faults. Thus, the knowledge about all faults that may happen in a DES is crucial from safety and economical point of view. Faults in DESs are modeled either explicitly by introducing fault events [7], [8], or indirectly by considering deviations from language specifications [9]. The ability to detect faults within a bounded time is called *fault diagnosability* [7], a property that must be valid for all faults according to the given DES model. This is a necessary condition to be able to construct a diagnoser for diagnosis of all faults in the model.

Since fault diagnosis is based on observers, the computation of a diagnoser has exponential complexity [1], while there are polynomial algorithms that can deduce about the diagnosability of a system [8], [10], [11]. However, although polynomial time algorithms exist, the state space increases exponentially when modular systems are composed. Thus, it is often too complex to analyze modular systems of industrial size. The evaluation of diagnosability for complete modular systems can, however, sometimes be simplified [12]–[14]. If, for instance, all subsystems are locally diagnosable, the total system is also diagnosable. Even if a modular system includes subsystems that are locally non-diagnosable, the total system may still be diagnosable [12].

One interesting approach to tackle the computational complexity is to apply abstraction based techniques to reduce the state space of a system, still keeping enough information to decide about diagnosability. Some recent results in this direction have been presented both for automata and PN models [9], [15], [16], including techniques for modular systems. In [9], the computational effort for diagnosability verification is reduced by determine sufficient conditions, such that diagnosability of the original system follows from diagnosability of an abstracted model. Moreover, it is shown that if the abstracted system is not diagnosable, then the original system is not diagnosable if all observable events remain after abstraction. This requirement implies that in general only limited abstractions can be expected for non-diagnosable systems. In [15], the problem is that the monolithic verifier must be built.

Opacity and anonymity verification With the rapid growth of communication networks and computer systems, cyber-security properties have attracted a lot

of attention. Opacity, as a security property [17], is dealing with hiding secrets from malicious observers [18], [19]. It is a general and formal property that has been widely investigated for DESs for finite automata [18], [20]–[22] and for PNs [23]–[25]. A system is opaque if, for any secret behavior, there exists at least one non-secret behavior that looks indistinguishable to the intruder [18], [26]. The security notion is investigated for automata [19] using either state-based predicates [18], [19], [25], [27], or language-based predicates [21], [24], [28]–[30].

There are different opacity notions, such as current-state, initial-state, and k-step opacity [19]. In [31], it is shown that these different types of opacities can be transformed to one another in polynomial time. This thesis focuses is on *current-state opacity* (CSO). A privacy notion that is adapted from CSO is *current-state anonymity* (CSA) [20], [30], which in [32] is used for location privacy. In this case, the servers that access the user's location information are considered as intruders.

An intruder with partial observation can be modeled as an observer of the system. There are several works that exploit observer generation for opacity verification [31], [33]–[36]. Given the exponential complexity of observer generation for verification purpose, as well as the complexity of interacting subsystems in modular systems of industrial size, state space explosion often occurs. Thus, reduction methods play an important role in making opacity and anonymity verification procedures feasible. In [37], a binary decision diagram technique [38] is used to abstract graphs to a moderate size for verification of three different opacity variants. In [39], a bisimulation-based method to verify the infinite-step opacity of nondeterministic finite transition systems is proposed. Since this abstraction is based on strong bisimulation, it has a minor reduction capability compared to abstractions where local events are hidden, such as weak bisimulation [40] and branching bisimulation (BB) [41]. Also, in [36], the conflict equivalence abstraction is used for opacity verification.

1.2 Research Questions

Based on the given background, different areas arise, for which further research is required. To set the boundaries of the topics to explore, the following research questions are to be answered:

RQ1 *How can polynomial diagnosability verification algorithms be used efficiently for both automata and Petri nets?*

RQ2 *How can a general abstraction method be formulated for verification of sys-*

tems, including both state and transition labels?

RQ3 *How can diagnosability verification be combined with incremental abstraction for modular systems?*

RQ4 *How can opacity and anonymity verification for modular systems be combined with incremental abstraction, especially when unobservable events are shared between different subsystems?*

1.3 Main Contributions

The following contributions are the result of the attempts to answer the research questions above.

- **(C1)** In Paper A [42], the first research question RQ1 is examined. First, the benefit of considering modular automata as PNs is demonstrated. A modified version of a basis reachability graph called MBRG is then used as an efficient reduction method. Based on the MBRG, which can be seen as a monolithic but reduced representation of a PN or modular automata, different polynomial diagnosability verifiers are compared. This includes an improvement of an existing verifier algorithm. A systematic evaluation on how complexity increases based on the number of sequences and tokens in PNs is presented. It is also demonstrated that theoretical complexity analysis, based on worst case scenarios, does not necessarily give the correct picture from a practical point of view.

- **(C2)** The modeling formalism that is used for verification of both diagnosability and opacity properties is called *transition system*, which includes both state and transition labels. In fault diagnosability verification, the corresponding necessary fault occurrence information is augmented to the states, and in opacity/anonymity verification the information regarding the non-safe states is stored as state labels.

  With this background, a general and flexible abstraction method called *branching bisimulation with state labels and explicit divergence* (BBSD) is defined in Paper B [43]. In Papers D [44] and E [45], this bisimulation is redefined, directly formulated as an equivalence relation and then called *divergent-sensitive visible* (DSV) bisimulation. This bisimulation preserves the temporal logic

properties that verify diagnosability, opacity, and anonymity. Indeed, it preserves CTL* except for the next operator, a fact that is also demonstrated on a model checking problem in Chapter 3. It is also shown in Paper D that the proposed abstraction approach, when being applied incrementally on a modular system, offers a significant state space reduction. The mentioned papers answer RQ2.

- **(C3)** In Papers B and C [46], the third research question RQ3 is presented. In these papers, the ultimate goal is to detect uncertain cycles, to be able to verify fault diagnosability. When applying incremental abstraction methods, these cycles may become silent and thus abstracted. Therefore, all uncertain cycles must be preserved during the abstraction, meaning that the abstraction must be *divergence-sensitive*. In Paper B, this is achieved by a generic incremental abstraction technique for modular systems based on BBSD.

  In Paper C, a new and simple transformation of a forbidden loop problem to a forbidden state problem is presented. This is done by introducing a simple detector automaton for each local forbidden cycle. The transformation to a nonblocking problem by introducing detectors is a generic technique that can be applied to a number of different verification problems, as in Paper E for security/privacy verification.

- **(C4)** The evaluation of research question RQ4 resulted in Papers D and E. An abstraction method for CSO verification of modular systems is proposed in Paper D, based on visible bisimulation [47]. The CSO verification problem is formulated as a temporal logic safety problem. The incremental abstraction is adapted to opacity verification, and it shows great computational time improvement compared to standard methods. A key factor in this incremental verification is that local observers can be generated before they are synchronized.

  Including also shared unobservable events, local observers cannot generally be computed. A combined incremental observer generation and abstraction for modular systems is then presented in Paper E. Some minor restrictions are introduced to be able to prove that the combined incremental observer generation and abstraction works correctly. This procedure includes additional temporary state labels, to avoid abstractions that may destroy the incremental observer generation.

1.4 Outline

This thesis is divided into two parts. The first part gives the reader an overview of the field of research, and a better understanding of the concepts developed in the papers that constitute the second part of the thesis.

The first introductory chapter provides the background, problem statement, as well as the research questions that have been the inspiration to the work and resulting contributions. In Chapter 2, preliminaries and basic notions that are used in this thesis are presented. Chapter 3 includes a general framework for modeling of modular discrete event systems, as well as verification and abstraction methods suitable for such systems. Chapter 4 focuses on fault diagnosability verification, and more specifically on incremental abstraction for diagnosability verification of modular systems. Moreover, in Chapter 5, it is shown how verification of some current state opacity and anonymity properties in modular systems can be performed in a modular and incremental framework. A summary of the appended papers is provided in Chapter 6, followed by some concluding remarks and directions for future research in Chapter 7.

CHAPTER 2

Preliminaries

In this chapter, some basic notions are introduced as a background for the rest of this thesis. The main modeling formalism is transition systems. However, since bounded Petri nets can represent modular automata and are used in Paper A, they are also shortly presented in this chapter.

2.1 Transition Systems

A discrete event model, including both state and transition labels, is often called a transition system [4], [48].

Definition 1 (**Transition system**)**:** A transition system G is defined by a 6-tuple

$$G = \langle X, \Sigma, T, I, AP, \lambda \rangle,$$

where

(i) X is a set of states,

(ii) Σ is a finite set of events,

(iii) $T \subseteq X \times \Sigma \times X$ is a transition relation, where a transition $t = (x, a, x') \in T$,

also denoted $x \xrightarrow{a} x'$, includes the source state x, the event label a, and the target state x',

(iv) $I \subseteq X$ is a set of possible initial states,

(v) AP is a set of atomic propositions, and

(vi) $\lambda : X \to 2^{AP}$ is a state labeling function. □

Automata and Kripke structures A transition system without state labels, where AP and λ are excluded from G, is an *automaton* without marked (final) states, also called a *labeled transition system* [49]. A transition system without transition labels (events), where Σ is excluded from G, results in a *Kripke structure*.

State labels Each state label includes a set of atomic propositions that are valid in the corresponding state. Typical state labels are marked and forbidden states, graphically denoted by double circles and crosses, respectively. Blocking states, from which it is not possible to reach a marked state, are examples of undesirable forbidden states. In fault diagnosability verification in Chapter 4, the state labels N and F are used to mark non-faulty and faulty states. In Chapter 5 on opacity and anonymity, the state label N denotes a non-safe state.

Transition function and active event set The transition relation T can alternatively be defined as a *transition function* $\delta : X \times \Sigma \to 2^X$, where the image is defined as $\delta(x, a) = \{x' \in X \mid x \xrightarrow{a} x' \in T\}$. Furthermore, the events that are involved in output transitions from a given state x, also called active or feasible events in state x, are included in the *active event set* $\Sigma(x) = \{a \in \Sigma \mid (\exists x' \in X)\, x \xrightarrow{a} x' \in T\}$. For a *deterministic* transition system, there is only one initial state, and for each state $x \in X$ and event $a \in \Sigma$, the number of elements in the image of the transition function is less or equal one, i.e. $|I| = 1$ and $(\forall x \in X)(\forall a \in \Sigma)\, |\delta(x, a)| \leq 1$.

Local and shared events In the composition of subsystems, see Def. 2, events that are included in no synchronization with other subsystems are called *local events*, while *shared events* are involved in more than one subsystem.

Modeling ε transitions The transition system G can be extended to include transitions, labeled by the empty string ε. The ε label is explicitly used for *local unobservable events*. Such events are not included in the alphabet Σ, while the total alphabet is extended to $\Sigma \cup \{\varepsilon\}$. A sequence of ε transitions $x = x_0 \xrightarrow{\varepsilon} x_1 \xrightarrow{\varepsilon}$

$\cdots \xrightarrow{\varepsilon} x_n = x'$, $n \geq 0$, is denoted by $x \overset{\varepsilon}{\Rightarrow} x'$. A corresponding sequence, including possible ε transitions before, after and in between events in a string $s \in \Sigma^*$, is denoted by $x \overset{s}{\Rightarrow} x'$. The *epsilon closure* of a state x is defined as $R_\varepsilon(x) = \{x' \mid x \overset{\varepsilon}{\Rightarrow} x'\}$, and for a set of states $Y \subseteq X$, we write $R_\varepsilon(Y) = \bigcup_{x \in Y} R_\varepsilon(x)$.

Projection A subset $\mathcal{L} \subseteq \Sigma^*$ is called a language. Moreover, for the event set $\Omega \subseteq \Sigma$, the natural projection $P : \Sigma^* \to \Omega^*$ is inductively defined as $P(\varepsilon) = \varepsilon$, $P(a) = a$ if $a \in \Omega$, $P(a) = \varepsilon$ if $a \in \Sigma \backslash \Omega$, and $P(sa) = P(s)P(a)$ for $s \in \Sigma^*$ and $a \in \Sigma$. Projections are especially used for unobservable local events.

Nondeterministic transition system A nondeterministic transition system generally includes a set of initial states, ε labeled transitions, and/or alternative transitions with the same event label. A transition function for an event $a \in \Sigma$ in a nondeterministic transition system is defined as $\delta(Y, a) = R_\varepsilon(\{x' \mid (\exists x \in R_\varepsilon(Y))\ x \xrightarrow{a} x' \in T\})$. An extended transition function is then inductively defined, for $s \in \Sigma^*$ and $a \in \Sigma$, as $\delta(I, sa) = \delta(\delta(I, s), a)$ with the base case $\delta(I, \varepsilon) = R_\varepsilon(I)$. Furthermore, the language for a nondeterministic transition system is defined as $\mathcal{L}(G) = \{s \in \Sigma^* | (\exists x \in I)\, \delta(x, s) \neq \varnothing\}$.

Local transitions and hidden τ events To obtain efficient abstractions, a special τ event label is used for transitions with *local observable events*. The lack of communication with other subsystems means that the τ event is hidden from the rest of the environment. The closure of τ-transitions in a finite path $x = x_0 \xrightarrow{\tau} x_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} x_n = x'$, $n \geq 0$ is denoted by $x \overset{\tau}{\Rightarrow} x'$.

Note the difference between ε and τ events. Unobservable local events are replaced by ε before an observer is generated, which removes any ε transitions. Observable local events are then replaced by τ to model that they are hidden, before performing any abstraction. In process algebra, the replacement of any specific event by the event τ is called *hiding*, cf. [40]. A transition system G where the events in Σ^h are hidden and replaced by τ is denoted by G^{Σ^h}.

Synchronous composition The definition of the synchronous composition in [50] is here adapted to τ events. Since the hiding mechanism only concerns local events, τ-labeled transitions are local. Thus, such events in different subsystems are not synchronized, although they share the same event label.

Definition 2 (Synchronous composition including τ events): Let $G_i = \langle X_i, \Sigma_i, T_i, I_i, AP_i, \lambda_i \rangle$, $i = 1, 2$, be two transition systems. The synchronous composition of G_1 and G_2 is defined as

$$G_1 \| G_2 = \langle X_1 \times X_2, \Sigma_1 \cup \Sigma_2, T, I_1 \times I_2, AP_1 \cup AP_2, \lambda \rangle$$

where

$$\begin{aligned}
(x_1, x_2) \xrightarrow{a} (x_1', x_2') \in T: \quad & a \in (\Sigma_1 \cap \Sigma_2) \setminus \{\tau\},\ x_1 \xrightarrow{a} x_1' \in T_1, \\
& \qquad\qquad\qquad\qquad x_2 \xrightarrow{a} x_2' \in T_2, \\
(x_1, x_2) \xrightarrow{a} (x_1', x_2) \in T: \quad & a \in (\Sigma_1 \setminus \Sigma_2) \cup \{\tau\},\ x_1 \xrightarrow{a} x_1' \in T_1, \\
(x_1, x_2) \xrightarrow{a} (x_1, x_2') \in T: \quad & a \in (\Sigma_2 \setminus \Sigma_1) \cup \{\tau\},\ x_2 \xrightarrow{a} x_2' \in T_2,
\end{aligned}$$

and $\lambda : X_1 \times X_2 \to 2^{AP_1 \cup AP_2}$. □

Modular systems A modular system consists of a number of interacting components or subsystems G_i, $i \in \mathbb{N}_n^+ = \{1, \ldots, n\}$. The interaction is assumed to be modeled by the synchronous composition, such that the total monolithic system is given by

$$G = \|_{i \in \mathbb{N}_n^+} G_i = G_1 \| G_2 \| \cdots \| G_n.$$

Although the number of states in the individual components $|Q_i|$ may be small, the total number of states in the synchronized monolithic system is in worst case $\Pi_{i=1}^{n} |Q_i|$. Furthermore, the observer, diagnoser or verifier generation adds more complexity, especially due to the exponential complexity in the observer generation. This fact strongly motivates the importance of utilizing reduction methods for modular systems.

2.2 Petri Nets

A place transition (P/T) net is a tuple $N = (P, T, Pre, Post)$, where P is a set of n_P places, T is a set of n_T transitions, $Pre : P \times T \to \mathbb{N}$ and $Post : P \times T \to \mathbb{N}$ are the pre and post incidence functions that specify the arcs between places and transitions. Furthermore, the incidence matrix is $C = Post - Pre$.

The marking vector $M : P \to \mathbb{N}$ assigns to each place of a P/T net a non-negative integer number of tokens. The marking of place p is denoted by $M(p)$. A Petri net (PN) $\langle N, M_0 \rangle$ is a P/T net N with an initial marking M_0. A transition t is enabled at M if $M \geq Pre(\cdot, t)$, and may fire, yielding the marking $M' = M + C(\cdot, t)$. We

write $M\,[\sigma\rangle$ to denote that the sequence of transitions $\sigma = t_{j_1} \cdots t_{j_k}$ is enabled at M, and $M\,[\sigma\rangle\ M'$ represents that the firing of σ yields M'.

Reachability graph A marking M is reachable in $\langle N, M_0\rangle$, if there exists a firing sequence σ such that $M_0\,[\sigma\rangle\ M$. The set of all markings reachable from M_0 defines the reachable set of $\langle N, M_0\rangle$, denoted $R\,(N, M_0)$. The reachability graph (RG) includes all reachable markings and the involved transitions. Since each value of the marking vector defines a state, this graph can also be considered as an automaton or a finite-state machine that is equivalent to the original PN.

Observable and unobservable transitions The set of transitions T is partitioned into the set of observable transitions T_o and the set of unobservable transitions T_u, such that $T = T_o \dot{\cup} T_u$. The set of fault transitions T_f is a subset of T_u, i.e. $T_f \subseteq T_u$. If there are r different fault classes, T_f can be partitioned into r different subsets T_f^i, where $i \in \mathbb{N}_r^+$.

Reduced model Among the set of transitions T, some transitions $T_a \subseteq T$ can be abstracted, resulting in an automaton that is smaller than the reachability graph RG. In [51], the concept of minimal explanations and basis markings are introduced, and a reduced basis reachability graph (BRG) is introduced for $T_a = T_u$. Thus, the resulting BRG only includes the observable transitions, but for diagnosability verification the unobservable fault events can not be abstracted. With $T_a = T_u \setminus T_f$ a modified basis reachability graph (MBRG) is achieved, which can be constructed without an exhaustive enumeration of the total state space [51]. This reduced MBRG automaton is used in Paper A as an abstraction method, from which different polynomial diagnosability verifiers are generated and compared.

Comparison between transition systems and Petri nets A modular DES can be represented as a set of synchronized automata, but also as one PN, where shared events executed synchronously in different subautomata are represented as one common transition in the PN model. In other words, a PN is able to explicitly show a concurrent behavior, thus being more readable than automata and transition systems, at least if the number of places and arcs are not too many. Some properties can also be analyzed by linear algebra, without generating an explicit state space representation, where the MBRG is such an example. On the other hand, there are more well-established abstraction methods for transition systems, which will be further discussed and illustrated in Chapter 3. Especially, the incremental abstraction proce-

dure that is also presented in Chapter 3 is a powerful reduction method for modular transition systems.

2.3 Observers

In this thesis, security/anonymity verification is based on observers that are achieved by subset construction [2]. There are several works that exploit observer generation for opacity verification [31], [33]–[35]. Observers are deterministic finite automata that determine the set of states that can be reached in a DES after an observable event has been executed. Within a state of the observer a number of unobservable events may be executed in the original DES. A diagnoser is a refined type of observer, where specific fault state labels are introduced after the occurrence of faults in the related DES.

Observer generation For a nondeterministic transition system G, where unobservable (local) events have been replaced by ε, a deterministic transition system with the same language as $\mathcal{L}(G)$, called an *observer* $\mathcal{O}(G)$, is generated by subset construction [2], where $\mathcal{O}(G) = \langle \widehat{X}, \Sigma, \widehat{T}, \widehat{I}, AP, \widehat{\lambda} \rangle$, and $\widehat{X} = \{Y \in 2^X \mid (\exists s \in \mathcal{L}(G))\, Y = \delta(I, s)\}$, $\widehat{T} = \{Y \xrightarrow{a} Y' \mid Y' = \delta(Y, a)\}$, and $\widehat{I} = R_\varepsilon(I)$. The relation between $\widehat{\lambda}(Y)$ and $\lambda(x)$ is application dependent, but the default assumption is that $\widehat{\lambda}(Y) = \bigcup_{x \in Y} \lambda(x)$. An obvious alternative is $\widehat{\lambda}(Y) = \bigcap_{x \in Y} \lambda(x)$, an interpretation that is applied in current state opacity. This topic is elaborated more in Chapter 5.

Introduce the transition function $\widehat{\delta}(Y, a) \stackrel{\text{def}}{=} \delta(Y, a)$ and the extended transition function, inductively defined as $\widehat{\delta}(\widehat{I}, sa) = \widehat{\delta}(\widehat{\delta}(\widehat{I}, s), a)$ with the base case $\widehat{\delta}(\widehat{I}, \varepsilon) = \widehat{I}$. It is then easily shown that $\widehat{\delta}(\widehat{I}, s) = \delta(I, s)$, see [2]. This means that $\mathcal{L}(\mathcal{O}(G)) = \mathcal{L}(G)$.

Example 1. Consider the system G and its observer $\mathcal{O}(G)$ in Fig. 2.1. The events $\{a, b, c\}$ are observable while the event u is unobservable. Replacing u with ε implies that the initial state in $\mathcal{O}(G)$ includes both state 0 and 2 in G, and the nondeterministic alternative when a is executed gives the second block state $\{1, 3\}$ in the observer. □

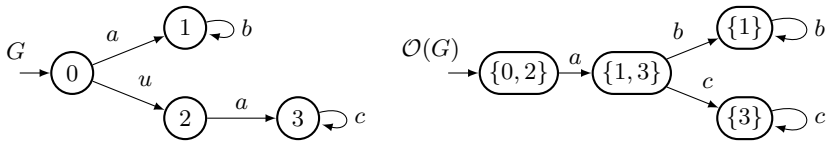

Figure 2.1: System model G and its observer $\mathcal{O}(G)$.

Online observer synchronization When all unobservable events are local, the observer of a modular system can be implemented as

$$\mathcal{O}(G) = \|_{i \in \mathbb{N}_n^+} \mathcal{O}(G_i).$$

This is shown in Paper E [45], for a general transition system. Thus, an observer of a modular system can then be implemented by running local observers combined with online synchronization. Possible transitions of the total observer $\mathcal{O}(G)$ are then determined by evaluating the current state and possible transitions of all local observers $\mathcal{O}(G_i)$. Hence, there is no need to explicitly generate $\mathcal{O}(G)$, but only the local observers and their current states.

This is a dramatic simplification compared to a traditional monolithic observer generation that may save huge amount of memory depending on the size of the total observer. Moreover, since building a diagnoser boils down to building an observer [52], this method can also be applied to the synchronization of local diagnosers, $\mathcal{D}(G) = \|_{i \in \mathbb{N}_n^+} \mathcal{D}(G_i)$.

CHAPTER 3

Incremental Abstraction

This chapter presents a general reduction method, based on an equivalence relation that preserves temporal logic system properties. The reduction method is utilized for verification of safety and security/privacy properties. Since the systems to be verified are assumed to be modular, an incremental reduction method is applied.

3.1 Temporal Logic

One of the verification methods is based on model checking, where it is evaluated whether a temporal logical specification is satisfied. This can be done automatically for transition systems, using a symbolic model-checker software tool such as *NuSMV* [53].

The two most well-known temporal logics are *linear temporal logic* (LTL) and *computation tree logic* (CTL). In LTL, temporal operators are provided for specification of properties along a single (linear) path, while in CTL operators quantify over the paths that are possible from a given state. The temporal operators in LTL are G = *always*, F = *eventually*, X = *next*, and U = *until*, all four ranging over the states along a particular path. Temporal operators in CTL are expressed in pairs, meaning that the operators G, F, X, and U are joined with path quantifiers as prefixes, either

the universal quantifier A or the existential quantifier E, which express *all paths* and *there exists a path*, respectively [54].

An example of an LTL formula is pUq, which specifies that for a given path, p holds in all states until q is valid. The CTL formula EFp specifies that there exists a path where eventually p holds.

Syntax and semantics of CTL* The extended computation tree logic CTL* combines LTL and CTL in the sense that a path quantifier can prefix an assertion composed of arbitrary combinations of the usual linear-time operators. The syntax of CTL* formulas is separated into a grammar for state formulas and a grammar for path formulas.

Definition 3 (Syntax of CTL*): State formulas in CTL* are defined inductively by the grammar

$$\psi ::= \mid \top \mid p \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \exists\varphi,$$

where p is an element in the set of atomic propositions AP, ψ, ψ_1 and ψ_2 are state formulas, and φ is a path formula. Path formulas in CTL* are defined inductively by the grammar

$$\varphi ::= \mid \psi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 U \varphi_2 \mid X\varphi,$$

where ψ is a state formula, and φ, φ_1 and φ_2 are path formulas. □

Additional ordinary propositional logic operators are derived operators, such as $\psi_1 \vee \psi_2 = \neg(\neg\psi_1 \wedge \neg\psi_2)$. The eventually and always operators are defined as $F\varphi = \top U \varphi$ and $G\varphi = \neg F \neg\varphi$, see further details in [4]. For a transition system, the semantics of CTL* is now formally defined.

Definition 4 (Semantics of CTL*): For a transition system $G = \langle X, \Sigma, T, I, AP, \lambda\rangle$, state formulas ψ, ψ_1, and ψ_2 are evaluated in states $x \in X$, while path formulas φ, φ_1, and φ_2 are evaluated along infinite paths ρ, starting in a state $x \in X$ at time instant t, and the path ρ^k denotes the suffix of ρ, starting at time instant $t+k$. The satisfaction relation $\models$ for different state and path formulas are here defined inductively as

$$\begin{array}{lll}
(1) & x \models \top & \Leftrightarrow \text{ always,} \\
(2) & x \models p & \Leftrightarrow p \in \lambda(x), \\
(3) & x \models \neg\psi & \Leftrightarrow x \not\models \psi, \\
(4) & x \models \psi_1 \wedge \psi_2 & \Leftrightarrow x \models \psi_1 \text{ and } x \models \psi_2, \\
(5) & x \models \exists\varphi & \Leftrightarrow \text{there exists a path } \rho \text{ such that } \rho \models \varphi,
\end{array}$$

$$
\begin{aligned}
(6)\quad & \rho \vDash \psi && \Leftrightarrow && x_0 = \text{first state in } \rho \text{ and } x_0 \vDash \psi, \\
(7)\quad & \rho \vDash \neg\varphi && \Leftrightarrow && \rho \nvDash \varphi, \\
(8)\quad & \rho \vDash X\varphi && \Leftrightarrow && \rho^1 \vDash \varphi, \\
(9)\quad & \rho \vDash \varphi_1 \wedge \varphi_2 && \Leftrightarrow && \rho \vDash \varphi_1 \text{ and } \rho \vDash \varphi_2, \\
(10)\quad & \rho \vDash \varphi_1 U \varphi_2 && \Leftrightarrow && \text{there exists a } k \geq 0 \text{ such that } \rho^k \vDash \varphi_2, \\
& && && \text{and for all } 0 \leq j < k, \rho^j \vDash \varphi_1.
\end{aligned}
$$

□

An example of a CTL* formula is the safety specification $EG(p \wedge q)$, which specifies that there exists a path where both p and q hold in every state. Another example is the liveness specification $AGFp$, which specifies that in all paths Fp holds in all states. This can also be expressed as "eventually p" is repeated forever, or in other words, that p has to be true infinitely often.

3.2 Visible Bisimulation Equivalence

In this section, a unified abstraction for temporal logic verification, called *visible bisimulation equivalence* is introduced. Equivalent states in the abstracted model have the property that if a CTL* formula is true in one state, it cannot be false in another equivalent state and vice versa. This is valid for all CTL* formulas, except for formulas including the *next* operator X. This temporal logic is called CTL*-X, and abstraction of a transition model based on visible bisimulation equivalence preserves all CTL*-X properties.

Bisimulation [40] defines a relation between the states of two models. The basic principle is that states are related if their next states are also related. Bisimulation relations between states are also proved to be equivalence relations. Two bisimulation relations that have a strong coupling to temporal logic are 1) *branching bisimulation* for labeled transition systems with only transition labels (no state labels), [41], [55], and 2) *stuttering bisimulation* for Kripke structures including only state labels [4].

In this thesis, these two formulations are unified in a bisimulation where both state and transition labels are included. Two other similar formulations that also include both state and transition labels are presented in [56] and [57]. They are, however, based on traditional relation based bisimulation formulations. The first one in [56] is called visible bisimulation. To emphasize that our alternative definition is directly formulated as an equivalence relation, it is in Paper D, [47] called *visible bisimulation equivalence*. All earlier bisimulation definitions are based on relations that are

shown to be equivalence relations, sometimes including complex proofs, especially for branching bisimulation [58]. To be able to directly apply our equivalence relation, the known properties for stuttering and branching bisimulation, especially the fact that all CTL*-X properties are preserved by this equivalence, are proved for this equivalence formulation in Paper D, [47].

The basic definition of our visible bisimulation equivalence is inspired by an algorithm for branching bisimulation called the signature algorithm [59]. However, that paper is based on the ordinary relation based branching bisimulation, and shows how the algorithm computes the expected relation based results. Our approach introduces a definition that directly generates the properties we are interested in. Thus, the theoretical analysis is significantly simplified by our definition. A detailed formulation and illustration of our visible bisimulation equivalence is presented in the rest of this section.

Partition Π and block $\Pi(x)$ To obtain reduced transition systems, states $x, y \in X$ that can be considered to be equivalent in some sense, denoted by $x \sim y$, are merged into equivalence classes $[x] = \{y \in X \,|\, x \sim y\}$, also called blocks. These blocks, which are non-overlapping subsets of X, divide the state space into the quotient set $X/\!\sim$, also called a partition Π of X. The block/equivalence class including state x is denoted by $\Pi(x) = [x]$. A partition Π_1 that is finer than a partition Π_2 means that $\Pi_1(x) \subseteq \Pi_2(x)$ for all $x \in X$. It is denoted by $\Pi_1 \preceq \Pi_2$.

Quotient transition system $G/\!\sim$ Blocks are the states in reduced transition systems, and the notion partition Π is used in the computation of this model, while the resulting reduced model takes the equivalence perspective. The reduced model is therefore called *quotient transition system*, and for a given partition Π it is defined as $G/\!\sim = \langle X/\!\sim, \Sigma, T_\sim, I_\sim, AP, \lambda_\sim \rangle$, where $X/\!\sim = \{[x] \,|\, [x] = \Pi(x)\}$ is the set of block states (equivalence classes), $T_\sim = \{[x] \xrightarrow{a} [x'] \,|\, x \xrightarrow{a} x'\}$ is the set of block transitions, $I_\sim = \{[x] \,|\, x \in I\}$ is the set of initial block states, and $\lambda_\sim([x]) = \lambda(x)$ is the block state label function, where it is assumed that $\lambda(x) = \lambda(y)$, $\forall y \in [x]$. The aim is to compute such reduced quotient transition systems, which still preserve relevant properties that can be analyzed, for instance, by temporal logic.

Invisible and visible transitions To obtain efficient reductions, the event τ is introduced as label for local transitions, included in no synchronization with other transition systems. The lack of communication with other models means that the τ event is *hidden* from the rest of the environment. Other events $a \neq \tau$ and cor-

responding transitions are then said to be *visible*. For a given state partition Π, a transition $x \xrightarrow{\tau} x'$ is *invisible* if $\Pi(x) = \Pi(x')$, while a transition $x \xrightarrow{a} x'$ is *visible* if $a \neq \tau$ or $\Pi(x) \neq \Pi(x')$. This means that a τ transition from one block to another is visible, while a visible event $a \neq \tau$ can be a self-loop or a transition from one block to another.

Divergent path, state, and block An infinite path $x = x_0 \xrightarrow{\tau} x_1 \xrightarrow{\tau} x_2 \xrightarrow{\tau} \cdots$ is *divergent* if all states in the path belong to the same block, i.e. , $x_i \in \Pi(x)$ for all $i > 0$. All states x_i included in a divergent path are *divergent states*, denoted by $x_i \hookrightarrow$. A block that only includes divergent states is called a *divergent block*.

Stuttering transition A path $x \xrightarrow{\tau} x_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} x_n \xrightarrow{a} x'$ is called a *stuttering transition*, denoted by $x \xrightarrow{a} x'$, if $\Pi(x) = \Pi(x_1) = \ldots = \Pi(x_n)$, and $a \neq \tau$ or $\Pi(x_n) \neq \Pi(x')$, meaning that the first n transitions are invisible, while the last one is visible.

Block transitions and event-target-blocks For a transition system with a transition $x \xrightarrow{a} x'$ and a partition Π, the corresponding block transition is denoted by $\Pi(x) \xrightarrow{a} \Pi(x')$, and the set of *event-target-blocks* for every state $x \in X$ is defined as

$$\Gamma_\Pi(x) = \{\xrightarrow{a} \Pi(x') \,\big|\, x \xrightarrow{a} x' \in T\}.$$

Obviously, $\Gamma_\Pi(x)$ includes all actual events $a \in \Sigma$ and target block states $\Pi(x')$ that are related to transitions $x \xrightarrow{a} x' \in T$ from the source state x. Corresponding set of *visible event-target-blocks* is defined as

$$\Gamma^v_\Pi(x) = \{\xrightarrow{a} \Pi(x') \,\big|\, x \xrightarrow{a} x'\}.$$

The element $\xrightarrow{a} \Pi(x')$ in $\Gamma^v_\Pi(x)$ represents a stuttering transition $x \xrightarrow{a} x'$, which indeed is a path including a set of invisible transitions within the block state $\Pi(x)$ followed by the last visible transition $x_n \xrightarrow{a} x'$. The final state x' belongs to a new block $\Pi(x') \neq \Pi(x)$ if $a = \tau$. If $a \neq \tau$, x' can also be a state in the current block, i.e. , $\Pi(x') = \Pi(x)$. In that case, the visible transition is a self-loop block transition $\Pi(x) \xrightarrow{a} \Pi(x') = \Pi(x)$.

Bisimulation Equivalences

Here, three specific partitions Π are introduced, with added complexity. All three guarantee that related states $x, y \in X$, belonging to the same block state (equivalence class), also have the same future behavior. The first case, called *strong bisimulation equivalence*, means that only those states x and y in the original model can be merged in a common block state $\Pi(x)$, where every transition $x \xrightarrow{a} x'$ is matched by a transition $y \xrightarrow{a} y'$ such that $y \in \Pi(x)$. Furthermore, the target block states must be equal, i.e. , $\Pi(x') = \Pi(y')$ and, therefore, $y' \in \Pi(x')$. In the second and third alternatives, transitions are relaxed by also accepting stuttering transitions, where the third case also takes care of divergent paths.

Definition 5 (Bisimulation Equivalences): Given a transition system $G = \langle X, \Sigma, T, I, AP, \lambda \rangle$, including both state and transition labels, a partition Π that satisfies the fixpoint

$$\Pi(x) = \{y \in X \mid \Pi \preceq \Pi_\lambda \wedge \Gamma_\Pi(x) = \Gamma_\Pi(y)\}, \tag{3.1}$$

where $\Pi_\lambda(x) = \{y \in X \mid \lambda(x) = \lambda(y)\}$, is

(i) a *strong bisimulation equivalence* when

$$\Gamma_\Pi(x) = \{\xrightarrow{a} \Pi(x') \mid x \xrightarrow{a} x'\},$$

and states $x, y \in \Pi(x)$ are strongly bisimilar, denoted by $x \sim y$,

(ii) a *visible bisimulation equivalence* when

$$\Gamma_\Pi(x) = \Gamma^v_\Pi(x) = \{\xrightarrow{a}\!\!\!\!\!\rightarrow \Pi(x') \mid x \xrightarrow{a}\!\!\!\!\!\rightarrow x'\},$$

and states $x, y \in \Pi(x)$ are visibly bisimilar, denoted by $x \sim_v y$,

(iii) a *divergence-sensitive visible (DSV) bisimulation equivalence* when

$$\Gamma_\Pi(x) = \Gamma^d_\Pi(x) = \Gamma^v_\Pi(x) \cup \{\hookrightarrow \mid x \hookrightarrow\},$$

and states $x, y \in \Pi(x)$ are divergence-sensitive visibly bisimilar, denoted by $x \sim_d y$. □

If states x and y are related as xRy, at the same time as their next states x' and y' belong to the same relation R, x and y are said to be strongly bisimilar. This well-known relation, which is also proven to be an equivalence relation, see [40], is easily

shown to generate the same partition as in Def. 5. The original relation is formulated either for automata or Kripke structures, see [4], while our formulation includes both state and transition labels in a unified framework. The visible bisimulation equivalence in Def. 5 corresponds to branching bisimulation for automata, and stuttering bisimulation for Kripke structures. These bisimulations are illustrated in the following example.

Example 1. Branching, stuttering, and visible bisimulation A number of transitions with hidden τ events, followed by a visible event a, can be simplified, as demonstrated for G_1 in Fig. 3.1. Introducing the block state $\{0, 1, 2\}$, the two hidden τ events are excluded in the corresponding quotient model, and only the visible a event remains. This is an example of a branching bisimulation for transition systems without state labels.

For Kripke structures, a similar reduction is possible. In that case, two or more consecutive states in a path, with the same state label, do not keep more information

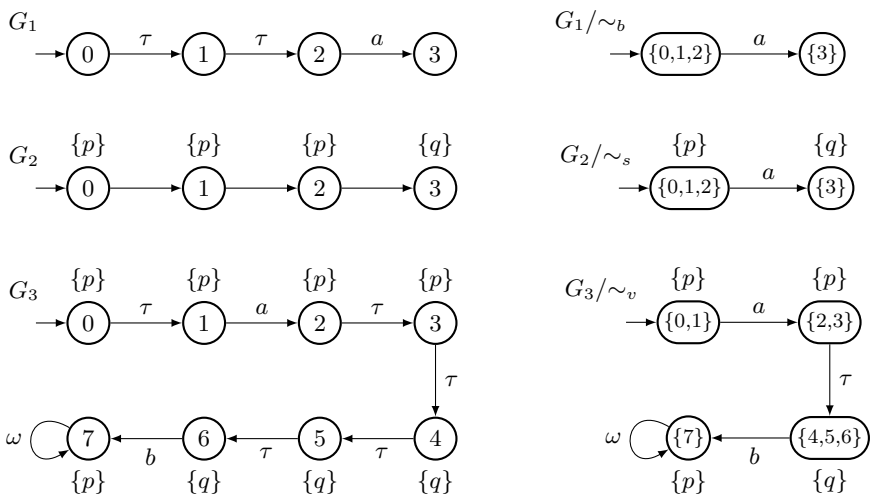

Figure 3.1: Transition system G_1 without state labels and its branching bisimulation quotient $G_1/\sim_b$, Kripke structure G_2 and its stuttering bisimulation quotient $G_2/\sim_s$, and transition system G_3 with both state and transition labels, and its visible bisimulation quotient $G_3/\sim_v$.

than one state with the same state label. This is illustrated in G_2 in Fig. 3.1, where the three first states with label $\{p\}$ are reduced to one block state $\{0, 1, 2\}$, followed by the final state with label $\{q\}$. Since transitions between states with the same state label are called stutter steps, this reduction is an example of a stuttering bisimulation.

Also consider G_3 in Fig. 3.1 that is a transition system including both state and transition labels. The state labels generate the state label partition $\Pi_\lambda = \{\{0, 1, 2, 3\}, \{4, 5, 6\}, \{7\}\}$, which, according to Def. 5 (ii), is not a visible bisimulation equivalence. However, the finer partition $\Pi = \{\{0, 1\}, \{2, 3\}, \{4, 5, 6\}, \{7\}\}$ is a visible bisimulation equivalence, since this is the coarsest partition where the resulting sets of event-target-blocks,

$$\begin{aligned}
&\Gamma^v_\Pi(0) = \Gamma^v_\Pi(1) = \{\overset{a}{\twoheadrightarrow} \{2, 3\}\},\\
&\Gamma^v_\Pi(2) = \Gamma^v_\Pi(3) = \{\overset{\tau}{\twoheadrightarrow} \{4, 5, 6\}\},\\
&\Gamma^v_\Pi(4) = \Gamma^v_\Pi(5) = \Gamma^v_\Pi(6) = \{\overset{b}{\twoheadrightarrow} \{7\}\},\\
&\Gamma^v_\Pi(7) = \{\overset{\omega}{\twoheadrightarrow} \{7\}\},
\end{aligned}$$

are equal within each block. The resulting bisimulation quotient $G_3/{\sim_v}$ is also shown in Fig. 3.1. □

In Example 1, strong bisimulation equivalence gives no reduction, i.e. , $G/{\sim} = G$. Moreover, according to the definition of visible bisimulation, τ events are only preserved in transitions to other blocks in the visible bisimulation quotient model.

Bisimulation computation of Π by fixpoint iteration Observe that (3.1) is a fixpoint, where each desired block $\Pi(x)$ depends on sets of *event-target-blocks* $\Gamma_\Pi(x)$, which in their turn depend on the blocks $\Pi(x)$ we are searching for. Furthermore, these blocks include all states $y \in \Pi(x)$ that satisfy the equality $\Gamma_\Pi(x) = \Gamma_\Pi(y)$ and Π is finer than Π_λ. Thus, this definition of $\Pi(x)$ determines the largest possible block states for the actual fixpoint. This corresponds to the coarsest partition of X, and the block states $\Pi(x)$ are obtained by solving (3.1) as the greatest fixpoint iteration

$$\Pi_{k+1}(x) = \{y \in X \,\big|\, \Pi \preceq \Pi_\lambda \wedge \Gamma_{\Pi_k}(y) = \Gamma_{\Pi_k}(x)\},$$

until $\Pi_{k+1}(x) = \Pi_k(x)$, see [60]. Then, no explicit state labels, corresponding to an automaton model, gives the default value $\lambda(x) = \varnothing$ for all $x \in X$, and $\Pi_\lambda = X$.

Quotient transition system Given a partition Π, the quotient transition system is defined on page 22. This applies to the strong bisimulation quotient $G/\!\sim$, the visible bisimulation quotient $G/\!\sim_v$, and the DSV bisimulation quotient $G/\!\sim_d$, with one exception. For $G/\!\sim_v$ and $G/\!\sim_d$, the transition relations are extended as

$$\begin{aligned} T_{\sim_v} &= \{[x] \xrightarrow{a} [x'] \,|\, x \xrightarrow{a} x' \wedge ([x'] \neq [x] \vee a \neq \tau)\} \\ T_{\sim_d} &= \{[x] \xrightarrow{a} [x'] \,|\, \big(x \xrightarrow{a} x' \wedge ([x'] \neq [x] \vee a \neq \tau)\big) \vee \\ &\qquad \big([x'] = [x] \wedge a = \tau \wedge x \hookrightarrow\big)\}. \end{aligned}$$

For the transition system G_3 in Example 1 and Fig. 3.1, the visible bisimulation quotient $G_3/\!\sim_v$ is shown in the same figure.

DSV Bisimulation Equivalence

Generally, τ self-loops are examples of divergent paths, which only include invisible transitions that stay within one block. We remind that all states x_i included in a divergent path are *divergent states*, denoted by $x_i \hookrightarrow$. Since the visible bisimulation equivalence neglects such divergent paths, this bisimulation equivalence is sometimes called *divergent-blind* visible bisimulation equivalence.

Divergent blocks and invisible cycles The DSV bisimulation equivalence preserves, according to Def. 5 (iii), the divergent behavior, by separating states such that either all states in a block are divergent, or no state in a block is divergent. The following example illustrates this phenomenon. Moreover, it also shows how the visible bisimulation equivalence can be adjusted to be able to also preserve the divergent behavior, i.e. , generating a DSV bisimulation quotient $G/\!\sim_d$. Observe that a cyclic path $x = x_0 \xrightarrow{\tau} x_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} x_n = x$ is an *invisible cycle*, denoted by $x\circlearrowleft$, if it is a divergent path, that is $x_i \in \Pi(x)$ for $0 \leq i \leq n$.

Example 2. Preserving divergent behavior by visible self-loops Consider the transition system G in Fig. 3.2, including an invisible cycle, the τ self-loop. The state label partition $\Pi_\lambda = \{\{0\}, \{1\}\}$, which means that no state reduction occurs in the visible bisimulation quotient $G/\!\sim_v$, and $\Pi = \Pi_\lambda$. However, according to Def. 5 for visible bisimulation, the τ self-loop in state 0 disappears in $G/\!\sim_v$, since only τ events in transitions to other blocks are preserved in the quotient model.

To solve this problem, a modified transition system $G^{\circlearrowleft}$ is formulated, where a visible $a_{\circlearrowleft}$ self-loop is added for each invisible cycle in G. In this example, an $a_{\circlearrowleft}$

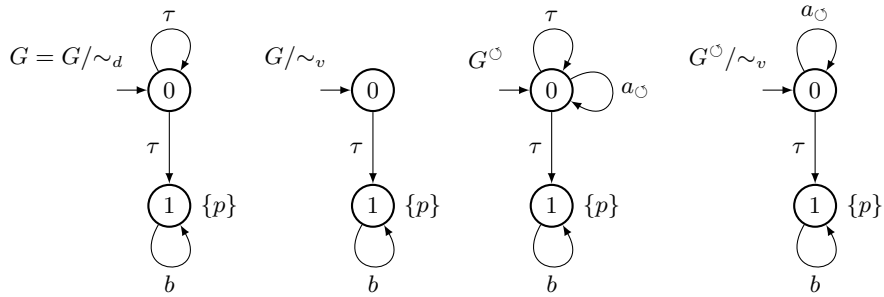

Figure 3.2: Transition system G that is equal to the DSV bisimulation quotient $G/\sim_d$, the visible bisimulation quotient $G/\sim_v$, the modified transition system $G^{\circlearrowleft}$ where a visible $a_{\circlearrowleft}$ self-loop is added to the invisible τ self-loop in G, and the visible bisimulation quotient $G^{\circlearrowleft}/\sim_v$.

self-loop is added in state 1, see Fig. 3.2. When the visible bisimulation equivalence is applied on $G^{\circlearrowleft}$, and finally the $a_{\circlearrowleft}$ event is replaced by τ, the DSV bisimulation quotient $G/\sim_d$ is achieved. □

DSV bisimulation by visible bisimulation with visible self-loops Consider two transition systems, $G = \langle X, \Sigma, T, I, AP, \lambda \rangle$ and the modified $G^{\circlearrowleft}$, where a visible self-loop $x \xrightarrow{a_{\circlearrowleft}} x$ is added in $G^{\circlearrowleft}$ for each invisible cycle $x\circlearrowleft$ in G. Then,

$$G/\sim_d = G^{\circlearrowleft}/\sim_v[\tau := a_{\circlearrowleft}].$$

This statement says that the DSV bisimulation quotient can be determined by computing the visible bisimulation quotient of $G^{\circlearrowleft}$, and then replacing the $a_{\circlearrowleft}$ event with τ. Example 2 and Fig. 3.2 confirm this procedure. Indeed, each invisible cycle can be computed by a depth-first search algorithm, and then directly be collapsed to a visible self-loop.

This method is a simplified alternative to adding a sink-state with a unique state label and a τ self-loop, see [4], [55]. Additional τ transitions from all divergent states in G are then also added to the sink-state. Our method utilizes the fact that we have both state and transition labels, while earlier methods are based on Kripke structures, where the event-based branching bisimulation is transformed to a stuttering bisimulation problem, see [55].

3.3 Abstraction of Modular Systems

To avoid state-space explosion in the synchronization of modular systems, an *incremental* abstraction technique is applied. Local events are then hidden and abstracted by the DSV bisimulation equivalence. As mentioned in the introduction of this section, this equivalence preserves all CTL*-X properties. Observe, however, that it is the divergent sensitive version that is required, to preserve all CTL*-X properties.

Incremental hiding and abstraction When subsystems are synchronized, more local events are obtained, which implies that more events can be hidden and abstracted. This hiding/abstraction method is repeated until all subsystems are synchronized. Utilizing this incremental technique, state space explosion is avoided when a reasonable number of events are local, or at least only shared with a restricted number of subsystems. Most real systems have this event structure, where still some events can be shared by all subsystems.

This incremental abstraction technique for modular systems can be traced back to [5], but its application to local events was more recently proposed in [6], where it was called compositional verification. In Papers B and C, this incremental abstraction is adapted to fault diagnosability verification, and in Papers D and E, it is adapted to opacity verification. In all cases, it shows great computational time improvement compared to standard synchronization without abstraction.

Generic algorithm The incremental abstraction algorithm outlined above is presented in Algorithm 1. The abstraction operator $\mathcal{A}$ in $G^{\mathcal{A}}_{\Omega_i}$ on line 6 also involves hiding, where local events are replaced by τ. The abstraction is only activated if there are any local events in G.

The basis of this algorithm is general, applicable for opacity and diagnosability, as well as temporal logic verification. However, for each verification problem, the corresponding suitable input system has to be generated, which includes n arbitrary subsystems, G_i, $i = 1, \ldots, n$, of a modular system. In fault diagnosability verification, the inputs are local verifiers $\mathcal{V}(G_i)$, and in opacity verification, the inputs are local observers $\mathcal{O}(G_i)$. All unobservable events must then be local, while shared unobservable events require some additional, but still minor, modifications of this algorithm. These modifications are presented in Chapter 5.

Different abstractions Algorithm 1 can be applied to any abstraction that is congruent with respect to synchronization and hiding. Examples of such abstractions are

Algorithm 1: Incremental abstraction

input $G_1, \ldots, G_n$
$\pi_\Omega := \{\{1\}, \{2\}, \ldots, \{n\}\}$
repeat
  Choose $\Omega_1, \Omega_2 \in \pi_\Omega$ according to some heuristics
  $\Omega := \Omega_1 \cup \Omega_2$
  $G_\Omega := G^{\mathcal{A}}_{\Omega_1} \parallel G^{\mathcal{A}}_{\Omega_2}$
  Replace Ω_1 and Ω_2 by Ω in π_Ω
until $\Omega = \mathbb{N}^+_n$
output $G^{\mathcal{A}}_\Omega$

Figure 3.3: Incremental abstraction of a modular transition system $G = \|_{i \in \mathbb{N}^+_n} G_i$.

DSV bisimulation equivalence that preserves CTL*-X properties and *conflict equivalence* [61] that preserves nonblocking. This algorithm was proposed for conflict equivalence abstraction in [6]. In Papers C and E, the efficiency of this abstraction is shown, where different verification problems are transferred to nonblocking verification problems.

Heuristics In the selection of the sets Ω_1 and Ω_2 and corresponding transition systems G_{Ω_1} and G_{Ω_2}, to be abstracted in Algorithm 1, a natural approach is to first select a group of transition systems with few transitions. Among them, the two systems with the highest proportion of local events are chosen to be abstracted. In this way, a significant reduction of states and transitions is achieved by the abstractions [6].

Example 3. DSV bisimulation quotient for a buffer-resource system Consider the transition system

$$G = R_1 \parallel B_1 \parallel R_2 \parallel B_2 \parallel R_3 \parallel Sp_3,$$

where the subsystems are shown in Fig. 3.4. R_1, R_2, and R_3 are resources that can be either in their idle (0) or busy (1) state, while B_1 and B_2 are buffers of size 2, located in between the resources. The last resource R_3 has two alternative choices when leaving the busy state. The choice could be to leave a part to one of two infinite buffers, modeled by the alternative events b_3 and c_3. The specification Sp_3 says that the alternative b_3 should be taken.

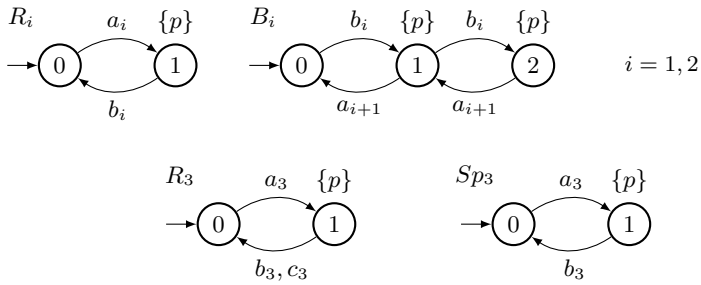

Figure 3.4: Transition system $G = R_1 \parallel B_1 \parallel R_2 \parallel B_2 \parallel R_3 \parallel Sp_3$.

The marked (goal) state of the total system G is the initial state $x = (0, 0, 0, 0, 0, 0)$, which is the only state where the state label is empty, i.e. $\lambda(x) = \varnothing$ and $x \models \neg p$. All other states have state label $\{p\}$, since the union of state labels of the subsystems becomes state label after synchronization.

The total abstracted transition system, denoted $G^{\mathcal{A}}$, is computed incrementally applying the DSV bisimulation equivalence. The initial step is $G^{\mathcal{A}}_{1,1} = (R_1^{\mathcal{A}} \parallel B_1)^{\mathcal{A}}$. This abstraction is based on the local events a_1 and b_1, followed by a_2 as additional local event in the abstraction $(G^{\mathcal{A}}_{1,1} \parallel R_2)^{\mathcal{A}}$. The event b_2 is the new local event in the next abstraction

$$G^{\mathcal{A}}_{1,2} = ((G^{\mathcal{A}}_{1,1} \parallel R_2)^{\mathcal{A}} \parallel B_2)^{\mathcal{A}}.$$

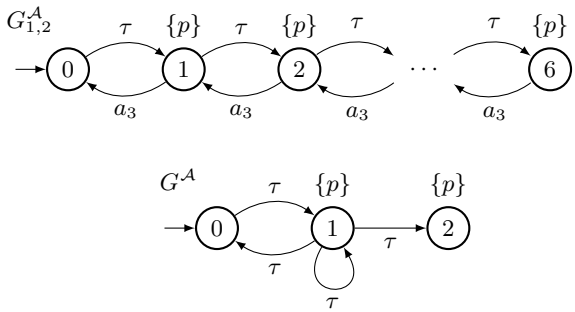

Figure 3.5: Abstracted transition system $G^{\mathcal{A}}_{1,2}$ and the final abstraction $G^{\mathcal{A}}$

In the last step, $(R_3 \parallel Sp_3)^{\mathcal{A}}$ is first computed, before the final abstraction $G^{\mathcal{A}} = (G^{\mathcal{A}}_{1,2} \parallel (R_3 \parallel Sp_3)^{\mathcal{A}})^{\mathcal{A}}$ is obtained. Fig. 3.5 shows that the intermediate transition system $G^{\mathcal{A}}_{1,2}$ has seven states, while the final abstraction $G^{\mathcal{A}}$ only has three states. Without abstractions G has 108 states. Even for an arbitrary number of buffers and resources, the number of states in the final abstraction is always three [47].

CTL*-X formulas can now be evaluated on the final abstraction $G^{\mathcal{A}}$. For instance

$$\begin{aligned} x &\vDash EF\neg p \quad \text{holds for } x = 0, 1, \\ x &\vDash AF\neg p \quad \text{does not hold for any state.} \end{aligned}$$

Since any CTL*-X property in G is also preserved in $G^{\mathcal{A}}$, this implies that the first formula also holds in the initial state of G, while the second formula also does not hold for any state in G.

The reason for the deadlock state in $G^{\mathcal{A}}$ is that the alphabet of the specification Σ_{Sp_3} does not include the event c_3, and R_3 can therefore execute this event as a local event without any synchronization with the rest of the system. To avoid this, the alphabet Σ_{Sp_3} must also include the event c_3. In that case, the deadlock state 2 in $G^{\mathcal{A}}$ disappears, but the formula $x \vDash AF\neg p$ still does not hold in any state, due to the τ self-loop in state 1. □

3.4 Model Checking and Incremental Temporal Logic Verification

The computational efficiency of the incremental abstraction in Algorithm 1, applying the visible bisimulation equivalence (VBE) and the divergent sensitive visible bisimulation equivalence (DVBE), will now be compared with some competitive alternatives. The implementation of the VBE and DVBE fixed points in Def. 5 are based on a minor adjustment and improvement of the routine SigRef [59], available in the open source software package mCRL2, www.mcrl2.org. This implementation is compared with the conflict equivalence (CE) algorithm, available in the software tool Supremica [62].

Since DVBE preserves CTL*-X, this abstraction is also compared with the symbolic model-checker nuXmv [63], which provides a number of efficient CTL and LTL verification algorithms. The default CTL and LTL algorithms in nuXmv are based on BDDs and therefore here called CTL_{BDD} and LTL_{BDD}. For verification of

LTL, a number of alternative more recent SAT-based algorithms are also available in nuXmv. The most efficient and robust alternative to LTL_{BDD} in our brief evaluation was found to be the algorithm k-liveness [64], in nuXmv and here called LTL_{IC3}.

The different algorithms are compared for two classical examples, Dijkstra's dining philosopher problem (DP) and a modified version of the buffer-resource system in Example 3. DPN includes N philosophers and N forks, a modular system including $2N$ subsystems, also available in Supremica [62]. Moreover, DPN:2 means that the first and the second half of the philosophers including forks are synchronized before they are delivered to the abstraction and verification routines. The number of states in the two subautomata is then $\max n_{s,i}$, which for DP8:2 means that the first abstraction has to act on a local model with $\max n_{s,i} = 1378$ states. The reason to evaluate this alternative model structure is to investigate the efficiency when larger subsystems are also included in a modular structure.

The second example is the buffer-resource system in Example 3, but here extended with one more buffer B_3 and one more resource R_4 that has the alternative event c_4 instead of R_3 in Example 3. All three buffers have the same size, either 20 and 50, and the models are called B20 and B50.

In Table 3.1, n_s is the number of states without abstraction, and $\max n_{s,i}$ is the number of states in the largest local model. The first three verfication methods (VBE,

Table 3.1: Execution times in seconds for nonblocking (VBE, CE, CTL_{BDD}) and liveness (DVBE, LTL_{IC3}, LTL_{BDD}) verification. Different dining philosopher (DPN) examples are evaluated, where N = number of philosophers. Also two buffer-resource (BM) models are examined, where M is the size of the buffers. The best execution times are given by bold numbers.

Model	n_s	$\max n_{s,i}$	VBE	CE	CTL_{BDD}	DVBE	LTL_{IC3}	LTL_{BDD}
DP4	431	7	**0.02**	0.03	0.09	**0.03**	0.27	0.09
DP4:2	431	66	**0.02**	**0.02**	0.10	**0.02**	3.62	0.12
DP6	9007	7	0.05	**0.03**	6.90	**0.08**	0.47	3.21
DP6:2	9007	302	**0.05**	0.13	2.74	**0.08**	16.6	2.35
DP8	$1.87 \cdot 10^5$	7	0.07	**0.06**	608	**0.19**	0.81	645
DP8:2	$1.87 \cdot 10^5$	1378	**0.25**	17.3	103	**0.37**	176	125
B20	$1.48 \cdot 10^5$	21	0.37	**0.12**	0.98	0.42	0.85	**0.15**
B50	$2.12 \cdot 10^6$	51	4.91	**1.01**	1.02	5.03	3.14	**0.73**

CE, CTL_{BDD}) evaluate nonblocking, which in CTL is expressed as $AG\,EF\,\varphi_m$, where φ_m holds in the marked (and in these examples the idle) state of the total system. In the last three columns (DVBE, LTL_{IC3}, LTL_{BDD}), liveness properties are evaluated. The LTL formula $GF\,\varphi_m$ (φ_m holds infinitely often) is evaluated in the DP examples. In the buffer-resource examples, the global deadlock states, marked by the state label φ_d, are avoided by the extended alphabet, including c_4 in the alphabet of Sp_4. The satisfaction of the LTL formula $FG\,\varphi_d$ is therefore verified in these examples.

The resulting execution times in seconds for the different abstraction and verification methods are given in Table 3.1, where the best nonblocking and liveness results are shown by bold numbers. Observe that the temporal logic formulas are trivially evaluated for the resulting abstracted models. For instance all abstracted DP models are the same as $G^{\mathcal{A}}$ in Fig. 3.5. To further evaluate the strength of VBE and DVBE compared to CE, some additional larger DP models are evaluated in Table 3.2. Based on the results in Tables 3.1 and 3.2, the following conclusions can be drawn.

1. When individual subsystems are large, VBE and DVBE are significantly better than all other evaluated methods. The reason why CE takes longer time for larger local models can be that this abstraction, although being a more efficient abstraction in terms of number of states, is more time consuming in each iteration compared to VBE.

2. For small subsystems CE is marginally better than VBE and DVBE. For the buffer example, CE is significantly better. But observe that CE can only eval-

Table 3.2: Execution times in seconds for VBE, CE, and DVBE, evaluated for different number N of dining philosophers DPN. The best execution times are given by bold numbers and O.T indicates out of time (>15 minutes).

Model	n_s	VBE	CE	DVBE
DP4	431	**0.02**	0.03	0.03
DP10	$3.9 \cdot 10^6$	**0.10**	0.17	0.42
DP20	$1.5 \cdot 10^{13}$	**0.22**	16.6	7.7
DP30	$6 \cdot 10^{19}$	**0.34**	O.T.	55.8
DP40	$2 \cdot 10^{26}$	**0.46**	O.T.	239
DP50	$9 \cdot 10^{32}$	**0.63**	O.T.	791

uate nonblocking, while DVBE preserves CTL*-X.

3. DVBE that also considers divergence-sensitivity, adds extra burden on the abstraction computation.

4. The BDD algorithms are sometimes efficient, but quite often they result in long execution times, typically when the evaluated model has many variables.

5. Although LTL_{IC3} is never the best choice, it is robust and sometimes even better than DVBE, but still sensitive to large submodels.

To summarize, the abstractions VBE and DVBE proposed in this thesis are often the most efficient choice for verification of temporal logic properties. However, further evaluations are necessary, and since VBE is sometimes significantly faster than DVBE, it is useful to further investigate for which temporal logic formulas it is enough to compute VBE without divergence sensitivity.

CHAPTER 4

Fault Diagnosability Verification

The behavior of DESs is partially observable through the execution of observable events, while the unobservable part also plays an important role in verification of safety properties. All DESs are prone to faults, and detection of faults as instances of *unobservable events* is crucial in fault diagnosis. Ensuring safety and reliability for complex modular systems is generally done by detection of faults and their isolation, which has received considerable attention in past years.

In this thesis we are focusing on *fault diagnosability verification* that is performed to decide a priori if all faults can be recognized within bounded time or not. One important problem in diagnosability verification of large modular systems is the complexity issue. A part of the work in this thesis is an attempt to reduce the complexity of modular DESs to verify diagnosability. The attempt is either related to finding polynomial algorithms for the diagnosability verification, or to find abstraction methods to reduce the size of the system. First, we present some notions that are necessary for fault diagnosability verification.

4.1 Faults and Diagnosers

Fault is an abnormal condition or a non-permitted deviation from acceptable behavior. A software bug, a broken sensor, or a short circuit are some instances of faults. In system modeling, depending on the problem, faults are usually considered as an additional input, and are assumed to be unobservable events. On the other hand, the termination of a system's ability to perform a desired task is often called a *failure*, for instance, when a system does not respond and gives wrong/unreasonable output [65].

The following example illustrates the difference in interpretation between these closely related notions. The result of a programmer's error is a fault in a written software, either in terms of a faulty instruction or data. When the program is running, the fault becomes active and produces an error. If and when the error affects a delivered service (in value and/or in timing of delivery), a failure occurs, cf. [66]. In some scientific areas this distinction is important [67], while in the DESs community, the two words "failure" and "fault" are used synonymously. In this thesis, we mainly use the notion fault.

Fault Diagnosis

There are two notions of diagnosis problem; online diagnosis and offline diagnosis, depending on whether the system to be diagnosed is in normal operation or not [7], [67].

Offline diagnosis Offline diagnosis means that the system is not in normal functioning and is in a test-bed [7]. For instance, in a repair shop, what a mechanics does to an automobile can be considered as offline diagnosis. This is also called the *diagnosability* problem, and refers to the ability to detect and identify faults within a finite delay after its occurrence. In diagnosability verification, the whole structure of the system and the complete characterization of the problem is available. Diagnosability verification is the basis of diagnosis, i.e., a fault can be diagnosed by a diagnoser only if the system is diagnosable with respect to a fault class. The diagnosability notion is an essential property that must hold when real life applications are constructed. However, even when a system is not diagnosable, a diagnoser may still diagnose some fault classes [68].

Online diagnosis In online diagnosis, or simply *fault diagnosis*, [12], [69]–[71], the system to be diagnosed is in normal operation. Hence, the operating state of

the system is constantly changing. In this case, the system cannot be opened for inspection, and the available measurements are limited to the observed system outputs. The whole diagnoser is not constructed, and the projection is generated only for the specific trace that is generated by the system [7].

Due to the significant differences between fault diagnosis and diagnosability verification, the corresponding approaches are also different [67]. Fault diagnosis means that a *diagnosis state* is associated to each observed string of events, where the states can be normal, uncertain, or indeterminate (faulty). On the other hand, diagnosability verification is about determining whether the system can detect all fault occurrences in a finite number of steps, cf. [8], [10], [72], which is the main focus of this chapter.

Diagnoser

Diagnosers are modified observers that carry fault information by augmented labels, attached to their states [7], [10]. One purpose of constructing diagnosers is to perform diagnosability verification. However, the state space of a diagnoser is in the worst case exponential in the cardinality of the state space of the system model [8]. It is more efficient if we could deduce about a system's diagnosability without having to construct a complete diagnoser. Fortunately, there are polynomial algorithms for this purpose, which are presented in the next section. However, the diagnoser construction is elaborated more to clarify the basis of the diagnosability notion.

N and F state labels For the diagnosis of DESs, diagnosers use observable events to detect and isolate faults. A *diagnoser* is constructed by traversing from possible initial states of a transition system, and augmenting all states with either N or F labels. A state label N is introduced, when no faulty transition is passed reaching to that state. Whereas, the state label F is introduced if there is at least one faulty transition on the way to reach to that state. As soon as a state label becomes F, all reachable states after that state are also augmented with F. Then, the diagnoser is obtained by generating an observer based on the extended state information. There can be different fault classes in a model, which can be identified with different indexes. However, in this thesis, without loss of generality, only one fault class in the system is assumed.

Example 1. The transition system G in Fig. 2.1 in Chapter 2, with the event u replaced by the alternative events u and f, is now labeled with state labels as G^L in Fig. 4.1. Here, the state labels are augmented to the state names. The events u and f are unobservable, where f is also a fault event. The diagnoser $\mathcal{D}(G)$ of the system

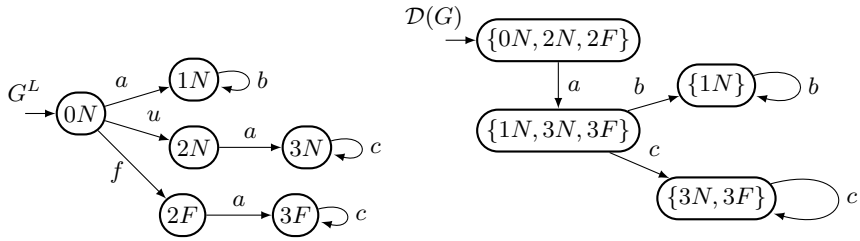

Figure 4.1: The labeled system model G^L and its diagnoser $\mathcal{D}(G)$.

is also depicted in the same figure, generated as the observer of G^L. There is a loop over state $\{3N, 3F\}$ that has different labels, and is called *uncertain state*. In the following, this is elaborated in detail. □

4.2 Fault Diagnosability Verification

When a system is designed, fault diagnosability verification is performed to decide a priori if all faults can be recognized within bounded time or not. This is performed on the whole system structure. Before defining the formal definition of fault diagnosability, the function that assigns labels to states is defined. A fault assignment function is a mapping from Σ to state fault labels N or F, i.e., $\psi : \Sigma \to \{F, N\}$. It means that if $\sigma \notin \Sigma^f$ (Σ^f is the set of faulty events), it is projected to N, otherwise it is projected to F. All reachable states after an F-labeled state, are also F-labeled.

Definition 6 (Diagnosability [8]): With respect to the event observation projection P and the fault assignment function $\psi : \Sigma \to \{F, N\}$, a system G is diagnosable if

$$(\exists n \in \mathbb{N})\,(\forall s, w \in \mathcal{L}(G))\,(\forall m = st \in \mathcal{L}(G)) :$$
$$\psi(s_f) = F \wedge |t| \geq n \wedge P(w) = P(m)$$
$$\Rightarrow (\exists r \in pr(\{w\}) : \psi(r_f) = F,$$

where s_f and r_f are the last events in the traces s and r, respectively, $|t|$ is the length of trace t, and $pr(\{w\})$ is the set of all prefixes of w. Also, $\mathcal{L}(G)$ is the set of all traces generated by G. □

This definition implies that a system is diagnosable, if it is possible to detect all

faults in the system within a bounded time, by observing a sequence of events. This means that the event observations after each fault in the system should be distinguishable enough compared to other observed sequences where no fault has happened.

Language specifications In order to uniquely identify each fault based on only observable events, the faulty behavior needs to be characterized. *Language specification* [9], [15], [69], [73] is one of the approaches that is utilized to represent the incorrect system behavior for fault diagnosability verification. In this approach, a specification represents the non-faulty behavior of the system and every deviation from that specification leads to a fault. There are some polynomial time algorithms that are developed to solve language diagnosability problems [69], [73], [74].

Explicit fault events Another approach for representing the unwanted system behavior is to include *fault events* [7], [75], where faults are shown as explicit events in the same model. Some of the polynomial algorithms for explicit fault-event diagnosability are presented in [8], [10], [11], [14], [75]. In this thesis, the focus is on this explicit fault-event approach, and Def. 6 is based on this faulty behavior characterization.

A common drawback of the mentioned approaches is that the complete state space of the system has to be enumerated, which is usually computationally infeasible. Therefore, many diagnosability verification methods exploit the system structure in order to avoid the explicit state representation of the overall system.

Decision Structures for Diagnosability Verification

There are three different structures for diagnosability verification. The first one is the *centralized* structure, where the diagnosis is calculated based on one *monolithic* diagnoser [7]–[10], [75]. In this structure, a centralized model is required to generate the centralized diagnoser. The computational complexity of this method is very high for large systems. However, to deal with geographically distributed systems or systems that are too large to be diagnosed with one single diagnoser, *decentralized* and *distributed* structures are developed. Moreover, system decomposition has been recognized as an important approach that mitigate the architectural complexity of systems.

Decentralized structure The *decentralized* architecture [69], [76]–[79] is close to the nature of a majority of technologically complex systems, such as communication and computer networks, manufacturing systems, and transportation systems.

These systems have several decision making sites that communicate with each other. These sites know the whole model, and use a local diagnoser, with similar functions and with possible communication between the local sites until reaching the same diagnosis as with a centralized diagnosis design. There is also a coordinator that decides about the final diagnosis. The main drawback is the ambiguity that may be in deciding about fault occurrences, and the way the local sites interact with each other. In [77] the notion of diagnosability is extended in order to better capture the decentralized architecture, including local sites communicating with a coordinator.

The notion of *codiagnosability* is introduced for decentralized diagnosers in [69], where it is required that the occurrence of any fault must be diagnosed within a bounded delay by at least one local diagnoser, using it's own observations of the system behavior. This diagnosability notion is also used in other works for both automata and PNs [11], [78], [80].

Distributed structure The last structure is the *distributed* structure [81]–[83], where the distinction between this structure and the decentralized structure is sometimes not clear. It was first considered as a special case of the decentralized structure in [77]. In this structure, in general, the local diagnosers merely use the local system models, which is different to the decentralized case. In other words, each subsystem only knows its own part of the global model and has local diagnosers to perform diagnosis locally. In distributed diagnosis approaches, each local diagnoser makes their diagnosis decision based on only a subset of observable events, and they communicate these decisions to other local diagnosers. The level of coordination required between the local diagnosers depends on how each local diagnoser is designed [84].

A *modular* structure [12], [73], [85] is a specific example of a distributed structure, where each fault in the system must be uniquely identified by the modular component where it occurs, and is solely based on event observations of that component [15]. This restriction reduces the computational complexity that is the major problem when dealing with DESs. Finally, note that diagnosability verifiers with polynomial complexity, which are presented in Sect. 4.3, can be adapted to the different structures mentioned above.

Uncertain and Indeterminate Cycles

Now that the diagnosability property is formally defined, in this subsection it is shown how it can be verified using diagnosers. Therefore, the notions of uncertain

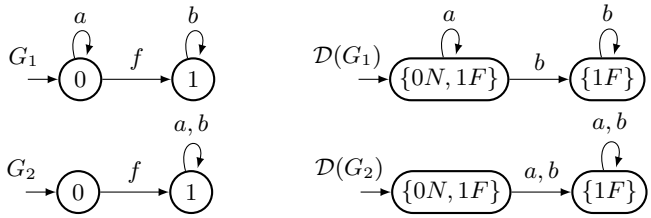

Figure 4.2: The diagnosable G_1 and its diagnoser $\mathcal{D}(G_1)$ that includes a loop over the uncertain state. The diagnosable G_2 and its diagnoser $\mathcal{D}(G_2)$ with no loop over the uncertain state.

and indeterminate cycles in diagnosers are clarified through some examples.

Example 2. Consider G_1 and its diagnoser $\mathcal{D}(G_1)$ in Fig. 4.2 with f as an unobservable fault event, and a and b as observable events. As it is seen in $\mathcal{D}(G_1)$, there is a loop with event a over the state that includes both F and N labels. This state is called an *uncertain state*, and a loop where all states are uncertain is an *uncertain loop*.

The diagnoser $\mathcal{D}(G_1)$ has one uncertain cycle, which corresponds to a cycle in G_1 with only label N. Thus, G_1 is diagnosable, since there is no other a loop in G_1 including label F. Therefore, traversing the a loop in $\mathcal{D}(G_1)$ does not violate the ability to diagnose the fault, because there is no ambiguity in the inverse projection from the a loop in $\mathcal{D}(G_1)$ to the a loop in G_1. In other words, to recognize whether a transition system is diagnosable or not, it is necessary to consider both the transition system and its diagnoser. First, find an uncertain cycle in the diagnoser, e.g. the a loop in $\mathcal{D}(G_1)$ in Fig. 4.2. Then, check the corresponding cycles in the transition system G_1, which is the inverse projection of that cycle in $\mathcal{D}(G_1)$. If there is a cycle in G_1, merely with either N or F labels, the corresponding cycle in $\mathcal{D}(G_1)$ is an uncertain cycle, and the system is diagnosable. The diagnoser $\mathcal{D}(G_2)$ depicted in Fig. 4.2 does not have any uncertain cycle. Thus, the system G_2 is also diagnosable. □

Indeterminate cycles In general, a cycle in a diagnoser is called an *indeterminate cycle*, if two cycles in the original system G, one including only label N and the other one with only F, can be associated with a cycle of uncertain states in $\mathcal{D}(G)$.

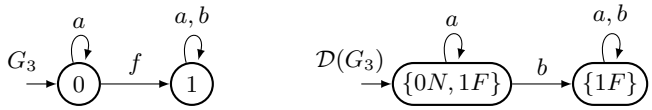

Figure 4.3: The non-diagnosable G_3 with an indeterminate cycle in its diagnoser $\mathcal{D}(G_3)$.

The presence of an indeterminate cycle implies violation of diagnosability.

Example 3. The diagnoser $\mathcal{D}(G_3)$ in Fig. 4.3, has one uncertain a loop that corresponds to two related cycles in G_3 depicted in Fig. 4.3. The first one is the a loop purely in state 0 where no error has occurred, and the second one includes a loops also in state 1, in which case the fault has occurred. Obviously, the diagnoser can not distinguish whether the system is in state 0 or state 1. In this case, the diagnoser contains an *indeterminate cycle*, and the system is not diagnosable. □

The following example illustrates further the importance of considering both system and its diagnoser together, when reasoning about the diagnosability of a transition system. Although diagnosers of two different systems can be identical, their corresponding systems may behave differently with respect to fault diagnosability.

Example 4. Consider Fig. 4.4, where both transition systems G_4 and G_5 contain an uncertain b loop in their diagnosers. Although, the diagnosers are identical, however, the b loop over the uncertain state, corresponds to only one b loop in G_4, while, there are two b loops in G_5. In G_5, state 1 is an N state, while state 4 is an F state. It means that there is an indeterminate cycle in $\mathcal{D}(G_5)$, and the system G_5 is therefore not diagnosable. □

The diagnoser generation has exponential complexity, which makes it infeasible to perform fault diagnosability verification for large DESs. To circumvent this problem different polynomial algorithms have been developed for diagnosability verification [8], [11], [86].

4.3 Diagnosability Verifiers

Fault diagnosis has been studied extensively for DESs since the early 90s. The basic concept is provided in [7], where a language and fault event based approach for online diagnosis and diagnosability verification is proposed, based on a diagnoser

construction. A similar state-based approach is later proposed in [75] for diagnoser construction, where an automata modeling formalism is proposed. Diagnosis has also been addressed by PN approaches in [72], [87].

The complexity of the method in [7] is exponential in the number of states and doubly exponential in the number of fault types. Other approaches, that have been developed later, address the complexity problem with polynomial algorithms, e.g., [8], [10], [11]. The diagnosability verification approach in [8] is automata-based, but it does not construct a diagnoser. A diagnosability verifier for a model G is then obtained by excluding the unobservable events, but still adding a state label L equal to N before, and equal to F after a fault has occurred, similar as in the generation of a diagnoser $\mathcal{D}(G)$. The resulting model G_o is synchronized with itself, such that a *verifier*

$$\mathcal{V}(G) = G_o \,\|\, G_o$$

is obtained.

Uncertain loops The verifier has states $(x_1\, L_1, x_2\, L_2)$, where there are four possible combinations of N and F state labels. Any state in the verifier that includes both an N and an F label is called an *uncertain state*, in the same way as a state in a

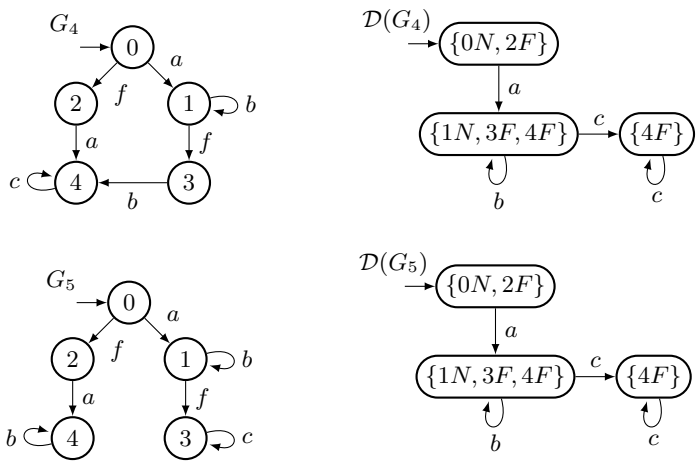

Figure 4.4: The diagnosable G_4 and its diagnoser $\mathcal{D}(G_4)$, and the non-diagnosable G_5 and its diagnoser $\mathcal{D}(G_5) = \mathcal{D}(G_4)$.

diagnoser is uncertain when it includes both N and F labels. If there are any loops with only uncertain states in $\mathcal{V}(G)$, the system G is not diagnosable. The reason is the two identical sequences of observable events in the two G_o models that reach to cycles, one including and the other one not including fault unobservable events. Since the faulty and the non-faulty behaviors can not be distinguished in finite time due to the cycles, the system is not diagnosable.

Polynomial complexity The computational complexity of this algorithm is polynomial and of 4th order in the number of states for G_o. Since all unobservable events are excluded, a great reduction is obtained when there are many unobservable events. An improved reduction to 2nd order in the number of states is obtained by a similar algorithm in [10] where, on the other hand, the unobservable events are not removed. Thus, the number of states is larger in the corresponding G_o model, especially when there are many unobservable events.

Synchronization of faulty and non-faulty model The method in [10] is further improved in [11], where symmetry is utilized such that one of the G_o models includes only the non-faulty part (only N labels). This is illustrated in the following example.

Example 5. The polynomial algorithm in [11] is utilized here to generate the verifier of G_4 that is depicted in Fig. 4.4 along with its diagnoser. Based on the verifier algorithm, states are first augmented with N and F labels, depending on if a fault has happened or not, as G_4^L in Fig. 4.5. The F-labeled states are then considered to be marked states. Backward reachability from all marked states is done, in order to get the faulty part G_4^F of the model. In this example, $G_4^F = G_4^L$. The non-faulty

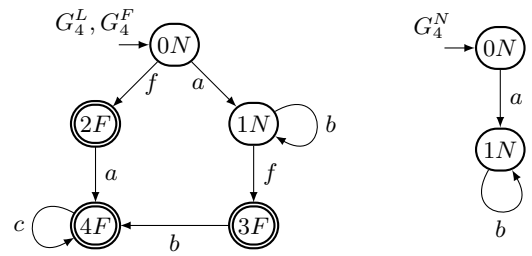

Figure 4.5: The faulty (G_4^F) and the non-faulty (G_4^N) parts of G_4.

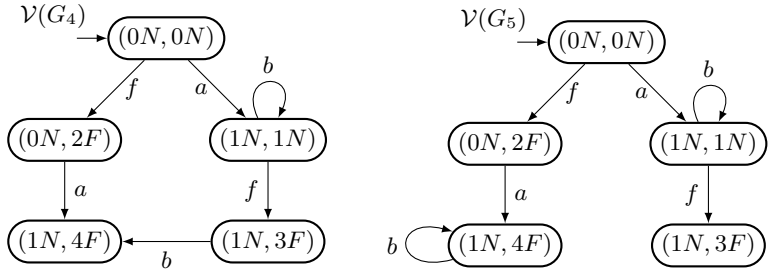

Figure 4.6: The verifiers $\mathcal{V}(G_4)$ and $\mathcal{V}(G_5)$.

part G_4^N, however, only has two states, and the verifier is obtained by computing $\mathcal{V}(G_4) = G_4^N \parallel G_4^F$, which is shown in Fig. 4.6. □

Example 6. Consider the transition systems and the diagnosers in Fig. 4.4. Although both systems have the same diagnosers, we have already observed that G_4 is diagnosable while G_5 is not. This is also confirmed in this example, by generating the verifiers of both systems, based on the algorithm in [11]. The faulty and non-faulty parts of G_4 are depicted in Fig. 4.5, and the verifiers for both G_4 and G_5 are shown in Fig. 4.6. As expected, $\mathcal{V}(G_4)$ does not have any loop over uncertain NF labeled states, meaning that G_4 is diagnosable, while $\mathcal{V}(G_5)$ has a loop over the uncertain state $(1N, 4F)$. Thus, G_5 is not diagnosable. □

In Paper A, a modified diagnosability algorithm is proposed that also reduces the state space by exploiting symmetry as in [11], while unobservable transitions are abstracted as in [8]. The notion of minimal explanations is also utilized, which is involved in MBRG [51], a graph for diagnosability analysis of PNs. Minimal explanations have a significant impact on complexity reduction, as they do not require an exhaustive enumeration of the state space. Considering modular automata as PNs, implies that the notion of minimal explanations also can be applied to automata-based diagnosability methods.

It is shown in Paper A that when there are a significant number of unobservable transitions, our proposed method gives better performance than [11] for systems with moderate size. Increasing the number of unobservable transitions results in an obvious increase in the size of the verifier in [11], while it does not change the number of states in our proposed verifier. However, for large systems, abstracting all unob-

servable transitions may generate a number of non-deterministic transitions in the verifier proposed in Paper A. This fact is shown to increase the state space of the verifier significantly for large systems.

Temporal Logic Specification

One possibility to verify fault diagnosis problems is to formulate a temporal logic expression that is verified by model checking algorithms [4]. Model checking techniques have been exploited for fault diagnosis of industrial systems in [88], [89]. In [90], examples of faults violating a formal specification expressed in temporal logic are presented, where LTL is used to express the fault specification. Also in [91], a temporal logic-based approach for diagnosing the occurrence of a repeated number of faults is developed. In [92], LTL is used as correctness requirement for fault detection of discrete-time stochastic systems. The temporal logic allows the correctness properties to be specified completely, and also supports automatic translation into automata. The work in [89] is on diagnosis of faults that recover once they occur, where the verification method is reformulated as an LTL formula.

CTL formula for uncertain loops In this thesis, CTL is used to specify diagnosability based on the verifier in [11]. The existence of loops over uncertain states in the verifier then needs to be analyzed, i.e. , if there are any loops with both N and F labels in each state. If there exists at least one such loop over NF states in the verifier, the system is not diagnosable.

Let the state formula Φ be true for uncertain states with NF labels. It means that Φ does not hold in *certain states* (with NN or FF labels in the verifier). A system is then not diagnosable if finally there exists at least one path where Φ is eventually always true in the verifier. In CTL, that is expressed as $EF\,EG(\Phi)$. Thus, a system is not diagnosable when $EF\,EG(\Phi)$ holds. On the other hand, since $EG(\Phi) \equiv \neg AF(\neg\Phi)$ and $EF(\neg\Phi) \equiv \neg AG(\Phi)$, we find that

$$EF\,EG(\Phi) \equiv EF\,(\neg AF(\neg\Phi)) \equiv \neg AG\,AF(\neg\Phi).$$

Thus, a system is diagnosable if $AG\,AF(\neg\Phi)$ holds, which means that all possible paths from all reachable states in the verifier will finally have no uncertain states.

Divergence

The *divergence* property [50] for DESs with partial observations implies that there are infinite sequences of transitions without any further interaction with the environment. In [93], two properties, *language divergence* and *marking divergence*, and their relation to diagnosability analysis of labeled PNs, are discussed. A related concept is loop-preserving observer for diagnosability verification in [9], which ensures that every loop in the original model also appear as loop in the abstracted model.

Divergence-sensitive abstraction In Sect. 3.2, visible bisimulation is introduced as an abstraction method for modular systems. Local events are then hidden by τ events and removed, as long as the abstracted system has an equivalent behavior. In its basic form, visible bisimulation removes any divergent loops only including τ events, while such loops are preserved in the divergence-sensitive version. Since diagnosability verification evaluates if there are any loops with uncertain states in the verifier, the divergence-sensitive version of visible bisimulation is critical. This is illustrated in the following example. In [43], [94], the divergence-sensitive version of visible bisimulation is applied to diagnosability verification of modular systems.

Example 7. The verifier for G_5 is depicted in Fig. 4.7. As it is seen, $\mathcal{V}(G_5)$ has a loop over the uncertain state $(1N, 4F)$, which means that G_5 is not diagnosable. However, when all events are local and have been replaced by τ, the loop over $(1N, 4F)$ will be removed by an abstraction that is not divergence-sensitive. Removing this τ loop makes the system diagnosable, which is not correct. Thus, it is important that any divergent loops are preserved in fault diagnosability verification,

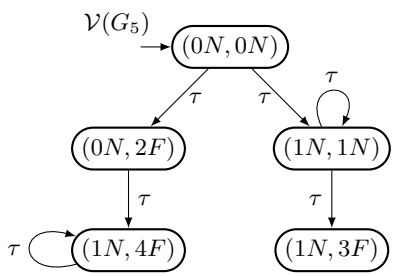

Figure 4.7: The verifier $\mathcal{V}(G_5)$, where the visible events in Fig. 4.6 have been replaced by invisible τ events.

which is guaranteed by a divergence-sensitive abstraction. □

The notion of divergence and an algorithm that preserves it, are described in Paper B, where dummy states are introduced as a trick in the abstraction procedure.

4.4 Diagnosability Verification for Modular Systems

Among all decision structures for diagnosability verification, monolithic diagnosability is considered in this thesis. However, it is considered for a general case, where the system G is modular and consists of n interacting subsystems, $G = \|_{i\in\mathbb{N}_n^+} G_i$, which are locally either diagnosable or non-diagnosable. All unobservable events Σ_i^u in the individual subsystems, including fault events, are assumed to be local. If all subsystems in a modular transition system are diagnosable, then the total system is diagnosable. However, if some subsystems are not diagnosable, the diagnosability of the total modular system needs to be verified [12], [13].

Modular Verifier

The polynomial algorithm presented in [11] and illustrated in Example 5, is now reformulated to also handle modular systems. According to the fault assignment function, states of each subsystem G_i are augmented with state fault labels from the set $\{N, F\}$, where the resulting transition systems are denoted by G_i^F. Then the corresponding non-faulty parts, G_i^N, are constructed, where all states labeled by $\{F\}$ in G_i^F, and their related transitions, are removed. In G_i^N, all unobservable events in Σ_i^u are also relabeled such that they are local in relation to G_i^F, but also to all other subsystems. The verifier $\mathcal{V}(G)$ is constructed by the synchronous composition of the two parts as

$$\mathcal{V}(G) = G^N \parallel G^F,$$

where $G^N = \|_{i\in\mathbb{N}_n^+} G_i^N$ and $G^F = \|_{i\in\mathbb{N}_n^+} G_i^F$. If the verifier $\mathcal{V}(G)$ contains at least one uncertain loop (cycle), it is not diagnosable. Also note that due to associative and commutative properties of the synchronous composition [1], the verifier can be rewritten as

$$\mathcal{V}(G) = (\|_{i\in\mathbb{N}_n^+} G_i^N) \parallel (\|_{i\in\mathbb{N}_n^+} G_i^F) = \|_{i\in\mathbb{N}_n^+} (G_i^N \parallel G_i^F) = \|_{i\in\mathbb{N}_n^+} \mathcal{V}(G_i). \quad (4.1)$$

This reformulation is important, since each pair $G_i^N \parallel G_i^F$ may have a number of local events. The modular abstraction, which is incrementally applied later, may then generate a significant state space reduction, before synchronization with additional subsystems is performed. This is explained in Chapter 3, and also further discussed in this chapter. In [11] the coreachability of G_F is also performed before the synchronization with G_N, which sometimes reduces the state space. However, it is not applicable in our modular version, and thus, the coreachability procedure is not included here.

State labels of verifiers The state labels that convey fault information may change during the synchronous composition of local verifiers in a modular framework. After every synchronization in (4.1), the union between the state labels, according to the definition of synchronous composition, only results in the two state labels N and $\{N, F\}$, independent of the number of synchronizations. In this thesis, these state label sets are simply denoted as N and NF, respectively.

Only non-diagnosable local verifiers The work in [14] is an adaptation of the algorithm in [11], where some minor improvements are suggested. There, the focus is on diagnosable systems where a local non-diagnosable behavior in one subsystem is blocked by another subsystem. For this reason, only the verifier of the non-diagnosable local subsystem is generated based on the polynomial approach in [11]. Then, the synchronous composition of the non-diagnosable verifiers with the rest of the original subsystems is performed. This approach shows whether the non-diagnosability on the subsystem level, can survive in the total modular system. In other words, this approach verifies if the NF loops in local non-diagnosable verifiers are preserved after being synchronized with the rest of the system or not.

Incremental Abstraction

Some recent works on abstractions applied to diagnosability verification have been presented both for automata and PN modeling formalisms [95]–[97], including techniques for modular systems [9], [15], [16], [43]. The idea of abstraction-based diagnosability for large-scale modular DESs is introduced in [9]. There, the computational effort for diagnosability verification methods is reduced by determining sufficient conditions, such that diagnosability of the original system follows from diagnosability of an abstracted model. Moreover, it is shown that if the abstracted system is not diagnosable, then the original system is not diagnosable if all observ-

able events remain after abstraction. This requirement implies that in general only limited abstractions can be expected for non-diagnosable systems. Diagnosability verification of modular systems with local specifications is considered in [15], where the same abstraction as in [9] is applied.

Bisimulation abstraction Although some minor improvements are suggested in the diagnosability algorithm in [14], more significant improvements are achieved by introducing an incremental abstraction of the verifier in (4.1). In Paper B, branching bisimulation including state labels is proposed as a generic incremental abstraction method, where local events are hidden and then abstracted such that temporal logic properties related to specific state fault labels are still preserved. The proposed algorithm can be seen as an instance of Algorithm 1 in Sect. 3.3 where the input models are the local verifiers $\mathcal{V}(G_i)$ in (4.1).

Transformation to a Nonblocking Problem

Since the diagnosability problem only includes two types of states, certain states with label N and uncertain states with label NF, the problem to detect any undesirable uncertain NF-cycles can be translated to a *nonblocking problem*. For this purpose, one arbitrary transition in each NF-cycle is labeled with a pair of events; its current event (σ) and an auxiliary unique event (w_j), i.e., $\langle \sigma, w_j \rangle$, for $j = 1, \ldots, m_i$, where m_i is the number of all NF-cycles in one single local verifier $\mathcal{V}(G_i)$. The resulting *local augmented verifier*, where such additional auxiliary events (w_j) are added, one for each NF-cycle, is denoted $\mathcal{V}_w(G_i)$.

Cycle detector For each NF-cycle $j = 1, \ldots, m_i$ in a local augmented verifier, $\mathcal{V}_w(G_i)$, a unique three-state automaton G_{ij}^D is introduced, which is called a *cycle detector*. A cycle detector for the j-th NF-cycle is shown in Fig. 4.8. The parallel composition of all such cycle detectors, $G_i^D = \|_{j \in \mathbb{N}_{m_i}^+} G_{ij}^D$, with $\mathcal{V}_w(G_i)$, results in the *extended local verifier*, $\mathcal{V}_e(G_i) = \mathcal{V}_w(G_i) \parallel G_i^D$.

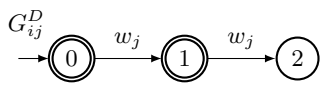

Figure 4.8: NF-cycle detector.

Extended verifier When the connected event $\langle \cdot, w_j \rangle$ in an NF-cycle has been executed two times in a verifier, one complete NF-cycle has been passed. At the same time, the cycle detector G_{ij}^{D} that is synchronized with that loop by the shared event w_j has moved from the initial to the non-marked blocking state 2. Thus, passing one forbidden NF-cycle generates a blocking state in the corresponding cycle detector. The total extended verifier is the parallel composition of all extended local verifiers. If any blocking state remains in the total extended verifier

$$\mathcal{V}_e(G) = \mathcal{V}_e(G_1) \parallel \mathcal{V}_e(G_2) \parallel \cdots \parallel \mathcal{V}_e(G_n),$$

this verifier has reachable blocking states, meaning that the original verifier $\mathcal{V}(G)$ includes at least one NF-cycle, and its corresponding modular system G is non-diagnosable.

Conflict equivalence abstraction In Paper C this extended modular verifier is reduced by incremental abstraction. The efficient conflict equivalence abstraction [61] that preserves nonblocking properties is then used in Algorithm 1 in Sect. 3.3. Thus, if $\mathcal{V}_e(G)$ after this abstraction is nonblocking, the system G is diagnosable, while a blocking abstracted verifier implies that G is not diagnosable.

CHAPTER 5

Opacity and Anonymity Verification

Some recent works have explored the analogy between opacity and diagnosability. Opacity can be related to the lack of diagnosability [98]. This relation is formally established in [30] using language-based opacity. In this chapter, the notion of *current state opacity* (CSO) and its adaptation to location-based security, called *current state anonymity* (CSA), are verified using incremental abstraction methods. The verification is mainly performed considering the existence of shared unobservable events in the system, followed by the special case where all unobservable events are local.

5.1 Background

Security and privacy concerns on the information flow of large communication networks and their diverse applications in modern technologies, are raised in recent years. It means that external observers, also referred as *intruders*, should not acquire the information flow in these services. There is huge amount of information exchanged on a daily basis between users, which unauthorized people may not have access to, and is referred to as *secret*. If for any secret behavior, there exists at least one indistinguishable non-secret behavior to the intruder, the system is called *opaque* [26].

The opacity notion is introduced for DESs, using PNs as the modeling formalism, first in [23] and then in [24]. It is also investigated for finite automata using either state-based or language-based predicates as in [18], [19], [25], [27] and [21], [28]–[30], respectively. In [31], the authors have shown that there exists a polynomial-time transformation between different notions of opacity to CSO. In CSO, the intruder is never certain that the current state of the system is within the set of secret states [18]. The intruder has full knowledge about the structure of the system. However, it only partially observes the system [98]. Therefore, it can be modeled as an *observer*.

Observer abstraction To perform CSO/CSA verification in a modular system, it is required to build the observer of the system. Given the exponential complexity of the observer generation, state space explosion often occurs while performing verification. Moreover, considering that a modular system consists of some interacting subsystems, it indicates a demanding computational complexity, especially when a large number of subsystems are interacting with each other. Therefore, abstraction methods can make it feasible to verify properties in complex modular systems. The work by [37] utilizes binary decision diagrams (BDDs) to abstract graphs of moderate size, as a method for the verification of three different opacity variants. Moreover, they prove that opacity properties are preserved by composition, which guarantees that local verification of these properties can also be performed. In [39], a method based on strong bisimulation is proposed to verify the infinite-step opacity of nondeterministic transition systems. The work in [36] also utilizes compositional abstraction for verification of opacity problems.

Incremental observer abstraction Our work in Paper D proposes an abstraction method for CSO verification of modular systems based on visible bisimulation abstraction [47]. In this method, both state and transition labels are integrated in the same abstraction. The advantage of this method is that temporal logic properties are preserved in the abstraction, and the opacity verification in Paper D is formulated as a temporal logic safety problem.

Abstractions of observers including shared unobservable events in modular systems is handled for the first time in Paper E. Here, the challenging part is that complete local observers can not be computed before some local models are synchronized. The reason is that shared unobservable events can not be reduced in the observer generation before they have become local after synchronization. In the same way, observable events are abstracted when they become local after synchronization. This means that the incremental abstraction in Algorithm 1 in Sect. 3.3 now must also

include an incremental observer generation, where unobservable events are removed when they become local. As a special case, when there are no unobservable shared events, the problem can be transformed into a nonblocking problem and an efficient abstraction can be incrementally applied for verification. This method is mentioned in Paper E, but also in Paper C for fault diagnosability verification.

5.2 Observer Generation in a Modular Framework

This section presents an incremental generation of reduced observers in a modular setting, applied to opacity verification. Differences and similarities between CSO and CSA are discussed, and shared unobservable events are included in the observer generation.

A straightforward approach for opacity verification of a modular system $G = \|_{i\in\mathbb{N}_n^+} G_i$ is to compute the observer of the monolithic system G. The drawback is then that incremental abstraction can not be utilized. However, when all unobservable events are local, complete local observers $\mathcal{O}(G_i)$ can be generated. This may be followed by incremental abstraction of the observable events, when they become local after synchronization of the local observers.

Combined abstraction and observer generation Shared unobservable events, on the other hand, can not be replaced by ε and removed in the observer generation, before they have become local after synchronization. At the same time, observable events should be abstracted as soon as they become local, to avoid state space explosion. The proposed solution is to extend the incremental abstraction with an incremental observer generation, meaning that an alternation between abstraction and observer generation can be performed when subsystems are synchronized. This alternation is repeated until all unobservable events are local and removed by observer generation. However, some precautions are required in this alternating procedure, to be able to accomplish local abstractions before the shared unobservable events are removed. This is handled by including some additional temporary state labels, which motivates for the more general and flexible visible bisimulation abstraction.

Incremental observer generation For a system $G = G_1 \parallel G_2$, the set Σ^ε includes all unobservable events in G that are replaced by ε. This set is a subscript in the observer operator $\mathcal{O}_{\Sigma^\varepsilon}(G)$. The sets of local unobservable events in G_1 and G_2

are Σ_1^ε and Σ_2^ε, respectively. The set Σ_{12}^ε includes the local unobservable events that appear after the synchronization of $\mathcal{O}(G_1) \parallel \mathcal{O}(G_2)$, which were previously shared between $\mathcal{O}(G_1)$ and $\mathcal{O}(G_2)$. Thus, $\Sigma^\varepsilon = \Sigma_1^\varepsilon \dot{\cup} \Sigma_2^\varepsilon \dot{\cup} \Sigma_{12}^\varepsilon$, and in Paper E it is shown that an observer for $G = G_1 \parallel G_2$ can be generated incrementally as

$$\mathcal{O}_{\Sigma^\varepsilon}(G_1 \parallel G_2) = \mathcal{O}_{\Sigma_{12}^\varepsilon}\big(\mathcal{O}_{\Sigma_1^\varepsilon}(G_1) \parallel \mathcal{O}_{\Sigma_2^\varepsilon}(G_2)\big). \tag{5.1}$$

It means that shared unobservable events are preserved, until they become local. The $\mathcal{O}_{\Sigma_{12}^\varepsilon}$ operator first replaces all local unobservable events belonging to Σ_{12}^ε by ε, and then generates the final part of the observer $O_{\Sigma^\varepsilon}(G_1 \parallel G_2)$. This result is illustrated in the following example.

Example 1. Consider the transition systems G_1, G_2, and $G_1 \parallel G_2$, as well as their observers in Fig. 5.1, where u and v are unobservable events and u is shared. This means that the sets of unobservable local event sets are $\Sigma_1^\varepsilon = \{v\}$, $\Sigma_2^\varepsilon = \varnothing$, and $\Sigma_{12}^\varepsilon = \{u\}$. Synchronization of the two models confirms the fact that $\mathcal{O}_{\{u,v\}}(G_1 \parallel G_2) \neq \mathcal{O}_{\{v\}}(G_1) \parallel \mathcal{O}_{\varnothing}(G_2)$, since the shared unobservable event u becomes local first after the synchronization. This requires one more observer generation $\mathcal{O}_{\{u\}}(\cdot)$ to obtain the final observer that is equal to $\mathcal{O}_{\{u,v\}}(G_1 \parallel G_2)$. This fact was recently also highlighted in [99]. □

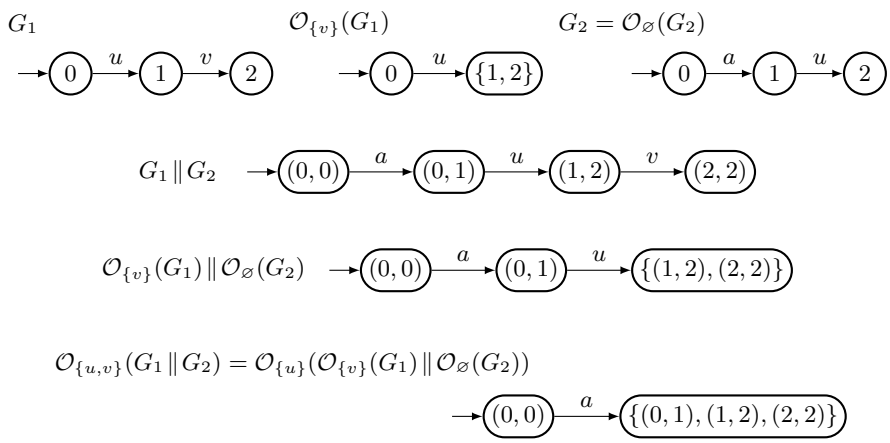

Figure 5.1: Transition systems G_1, G_2, and $G_1 \parallel G_2$, as well as their observers.

Incremental abstraction and observer generation In the same way as local unobservable events, included in a set Σ^{ε}, are replaced by ε and then are removed in the observer generation, *local observable events* in a system G, included in a set Σ^{h}, are *hidden* by replacing them with the event τ, resulting in the system G^{Σ^h}. This hiding, followed by the generation of a quotient transition system, results in the *abstracted transition system* $G^{\Sigma^h}/\!\sim \stackrel{\text{def}}{=} G^{\mathcal{A}^{\Sigma^h}}$. This also means that G^{Σ^h} is equivalent to $G^{\mathcal{A}^{\Sigma^h}}$, denoted by $G^{\Sigma^h} \sim G^{\mathcal{A}^{\Sigma^h}}$.

Every abstraction equivalence $\sim$ that is congruent with respect to synchronization and hiding can be applied, but the system is assumed to include state labels. The abstraction applied in this chapter is mainly the divergence sensitive visible bisimulation equivalence that preserves CTL*-X properties, see further details in Chapter 3.

Now, let Σ^h be partitioned in the same way as Σ^{ε}, as $\Sigma^h = \Sigma^h_1 \dot{\cup} \Sigma^h_2 \dot{\cup} \Sigma^h_{12}$. In Paper E, it is then shown that an incremental abstraction combined with observer generation can be formulated as

$$\mathcal{O}_{\Sigma^{\varepsilon}}(G_1 \,\|\, G_2)^{\mathcal{A}^{\Sigma^h}} \sim \mathcal{O}_{\Sigma^{\varepsilon}_{12}}\big(\mathcal{O}_{\Sigma^{\varepsilon}_1}(G_1)^{\mathcal{A}^{\Sigma^h_1}} \,\|\, \mathcal{O}_{\Sigma^{\varepsilon}_2}(G_2)^{\mathcal{A}^{\Sigma^h_2}}\big)^{\mathcal{A}^{\Sigma^h_{12}}}. \tag{5.2}$$

The observable and unobservable events are incrementally replaced by τ and ε, respectively, when they become local. The mix between step-wise abstraction and observer generation means that the observer generation requires some restrictions in the incremental abstractions. This is solved by introducing some additional temporary state labels that are presented in Sect. 5.3.

Current State Opacity and Anonymity

In CSO, the observer states that exclusively include secret states are called *non-safe*, while in CSA, the singleton observer states are considered as non-safe. The aim is, therefore, to verify the existence of non-safe states in a system observer to conclude about its opacity/anonymity. To augment the information regarding the non-safe states to the local observers, the state label N is introduced and preserved during abstraction. Here, a centralized architecture is considered, which includes one single intruder that has full knowledge of the system structure. However, the intruder can only observe a subset of the events, the observable events. In the following, local and monolithic observers for both CSO and CSA verification of modular systems are defined, and the differences in their state labels are explained.

Current state opacity In CSO, the aim is to evaluate if it is possible to estimate every secret states in a system, based on its observable events. For a transition system G, let $X_S \subseteq X$ be the set of secret states. This system is called *current-state opaque* if for every string of observable events $s \in \mathcal{L}(G)$, each corresponding block state $Y = \delta(I, s)$ in the observer $\mathcal{O}(G)$ that includes secret states also includes at least one non-secret state from the set $X \setminus X_S$. Such states are called *safe* states.

Thus, G is *current-state non-opaque* if and only if at least one block state in the observer $\mathcal{O}(G)$ includes only secret states from G, and is therefore *non-safe*. The label N is a state label for all non-safe states in $\mathcal{O}(G)$.

Current state anonymity With the increasing popularity of location-based services for mobile devices, privacy concerns about the unwanted revelation of user's current location are raised. For this reason, the notion of CSO is adapted, and the new notion called CSA is introduced in [32], which captures the observer's inability to know for sure the current locations of moving objects. A system G is *current state anonymous* if for all strings $s \in \mathcal{L}(G)$, the size of all corresponding block states $Y = \delta(I, s)$ in the observer $\mathcal{O}(G)$ is not a singleton ($|Y| > 1$). Such states are called *safe* states.

Thus, G is current-state non-anonymous, if and only if at least one block state Y in the observer $\mathcal{O}(G)$ is a singleton and is therefore *non-safe*. This is natural, since more than one system state in each observer block state implies an uncertainty in determining the exact location of a moving object. Finally, in the same way as for CSO, the label N is a state label for all non-safe (non-anonymous) states in the observer $\mathcal{O}(G)$.

The following example shows the observers for CSO and CSA, including their different N (non-safe) state label interpretations.

Example 2. Consider the transition system G in Fig. 5.2 where the secret state set $X_S = \{0, 1\}$. The event u is unobservable and is therefore replaced by ε before the observer generation. Although the observers are structurally equal, depending on the verification problem, the interpretation differs concerning the non-safe states and therefore, the state labeling. The block state $\{1\}$ in the observer $\mathcal{O}_{\text{CSO}}(G)$ has label N, because state 1 is a secret state. On the other hand, both states 1 and 2 in $\mathcal{O}_{\text{CSA}}(G)$ have state label N, since both are singleton states.

Since G includes no subsystems, all events can be considered as local. The observable events a and b are, therefore, hidden and are relabeled with τ. In the visible bisimulation (VB) abstraction in $\mathcal{O}_{\text{CSO}}(G)^{\mathcal{A}^{\{a,b\}}}$, there is no reduction, while in

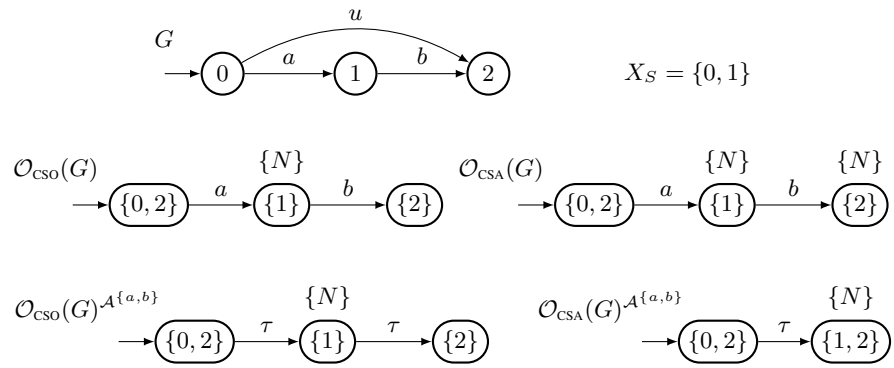

Figure 5.2: A system model G, its two observers $\mathcal{O}_{\text{CSO}}(G)$ and $\mathcal{O}_{\text{CSA}}(G)$ when the event u is unobservable, and their corresponding VB abstractions $\mathcal{O}_{\text{CSO}}(G)^{\mathcal{A}^{\{a,b\}}}$ and $\mathcal{O}_{\text{CSA}}(G)^{\mathcal{A}^{\{a,b\}}}$ when a and b are local observable events.

$\mathcal{O}_{\text{CSA}}(G)^{\mathcal{A}^{\{a,b\}}}$, both N-labeled states are merged. □

To summarize this part, a block state Y in the observer $\mathcal{O}(G)$ is *non-safe* and is augmented with state label N, in CSO verification when $Y \subseteq X_S$, and in CSA verification, when $|Y| = 1$. These results are now generalized to modular systems.

CSO and CSA verification for modular systems For a modular system $G = \|_{i\in\mathbb{N}_n^+} G_i$ we assume first that there are no shared unobservable events. Then, the observer can be expressed as the composed observer $\mathcal{O}(G) = \|_{i\in\mathbb{N}_n^+} \mathcal{O}(G_i)$ including block states $Y = (Y_1, \ldots, Y_n)$ that are *non-safe* and augmented with state label N, if in

- CSO verification, at least one of the local block states Y_i is non-safe and augmented with N. Thus, the label of the block state $Y = (Y_1, \ldots, Y_n)$ is the *union* of the labels of the local block states Y_i, $i \in \mathbb{N}_n^+$.
- CSA verification, all block states Y_i are non-safe and augmented with N. Thus, the label of the block state $Y = (Y_1, \ldots, Y_n)$ is the *intersection* of the labels of the local states. A state is, therefore, non-safe if all elements Y_i are singletons.

In the case of shared unobservable events, CSA verification is the same as above. For CSO, on the other hand, the definition of secret states becomes more complex when

shared unobservable events are involved, and we refer to Paper E for more details, including a deeper discussion on possible practical interpretations of these abstract formulations.

5.3 Incremental Observer Abstraction in the Presence of Shared Unobservable Events

When incremental abstraction is combined with incremental observer generation due to shared unobservable events, some restrictions must be included in the abstractions. Example 3 illustrates that abstraction before observer generation may generate a wrong result. Before this example, two important remarks are given.

(i) Initial local observers are always assumed to be generated before every hiding and abstraction. This means that, to simplify the notation, every transition system G is by default an observer, although not explicitly expressed. Thus, G is assumed to be deterministic, and all non-safe states are labeled by N.

(ii) Hiding and abstraction are always performed on deterministic systems. Hence, alternative choices, including τ events after hiding, are interpreted as deterministic choices in observer generation. Restrictions will also be included such that repeated observer generation and abstraction still means that G can be regarded as a deterministic transition system, although it may include alternative choices involving τ events.

Example 3. A deterministic system G is shown in Fig. 5.3, where the non-safe states are labeled by N and the event u is unobservable, while the event a is observable. The transition system $\mathcal{O}_{\{u\}}(G)^{\mathcal{A}^{\{a\}}}$ is obtained when the CSO observer generation is first performed, with u replaced by ε, followed by the abstraction where $\{a\}$ is replaced by τ. When the opposite order is applied, the transition system $\mathcal{O}_{\{u\}}(G^{\mathcal{A}^{\{a\}}})$ is obtained. Since abstraction can always be made after observer generation, the first result is correct. Abstraction before observer generation means, in this example, that states 1 and 2 are incorrectly merged, while the observer merges states 0 and 1, and the CSO rule says that a block state with both non-safe and safe states results in a safe state without state label N. □

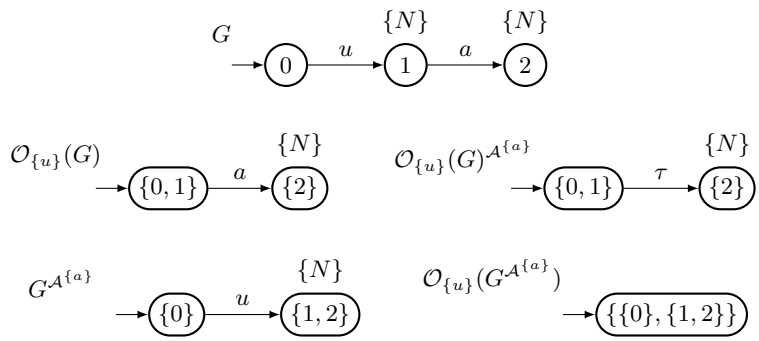

Figure 5.3: Different orders of observer generation and abstraction for transition system G.

Restrictions before Abstraction

In an alternating abstraction and observer generation procedure, Example 3 shows that it is important to include some restrictions, when abstraction is performed before observer generation. Two specific restriction rules are introduced to solve this problem.

Unique state labels before and after unobservable events For a system including a set Σ^{uo} of unobservable events, the first restriction is to introduce temporary unique state labels to the source and target transitions of every transition with event label in Σ^{uo}. These states are called Σ^{uo} source and target states (STSs). Adding unique state labels for all Σ^{uo} STSs means that VB abstraction does not influence future observer generations in a negative way.

Example 4. The same transition system as in Example 3 is considered, where unique state labels λ_s^u and λ_t^u are added to the source and target states of the transition $0 \xrightarrow{u} 1$. These STS labels prevent the VB abstraction $G^{\mathcal{A}^{\{a\}}}$ to merge the states 1 and 2, since they have now different state labels. The succeeding CSO observer generation $\mathcal{O}_{\{u\}}(G)^{\mathcal{A}^{\{a\}}}$ merges the source and target states of the transition $0 \xrightarrow{u} 1$, and the obsolete Σ^{uo} STS labels are removed. The resulting observer $\mathcal{O}_{\{u\}}(G)^{\mathcal{A}^{\{a\}}}$ in Fig. 5.4 now coincides with the correct observer $\mathcal{O}_{\{u\}}(G)^{\mathcal{A}^{\{a\}}}$ in Fig. 5.3. □

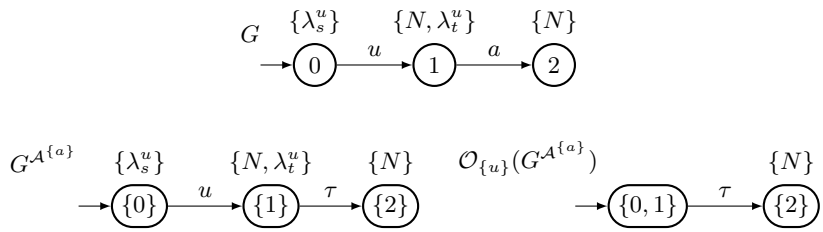

Figure 5.4: Unique Σ^{uo} STS labels λ_s^u and λ_t^u added to G, resulting in a correct observer $\mathcal{O}_{\{u\}}(G^{\mathcal{A}^{\{a\}}})$, where VB abstraction is performed before observer generation.

Future nondeterministic choices As mentioned in the beginning of this section, any model that is abstracted can be assumed to be deterministic due to an initial observer generation. After some synchronizations, however, additional local unobservable events may appear that are also replaced by ε. Additional nondeterministic choices may then generate deviations depending on the order between abstraction and observer generation. The following example illustrates this phenomenon.

Example 5. Consider the system G in Fig. 5.5, in which the unobservable event u has become local after some synchronizations, and therefore has not been earlier removed by observer generation. At this stage, only b is assumed to be shared, while all other events are assumed to be local. The observer is generated for CSO. Since state 6 in G does not have label N, the block state $\{2, 6\}$ in $\mathcal{O}_{\{u\}}(G)$ is safe (no label N). In the same way, since state 6 in $G^{\mathcal{A}^{\{a,c,d\}}}$ does not have label N, the block state $\{2, 3, 4, 6\}$ in $\mathcal{O}_{\{u\}}(G^{\mathcal{A}^{\{a,c,d\}}})$ is also safe.

Obviously, this is not a correct result, since the correct abstracted observer $\mathcal{O}_{\{u\}}(G)^{\mathcal{A}^{\{a,c,d\}}}$, where the abstraction is taken after the observer generation, includes the non-safe block state $\{3, 4\}$. This is not included in $\mathcal{O}_{\{u\}}(G^{\mathcal{A}^{\{a,c,d\}}})$, since the abstraction has incorrectly joined the states $2, 3$, and 4 before the observer generation. The reason for this problem is the nondeterministic choice that does not exist from the beginning, but appears later when the unobservable event u becomes local. □

This example illustrates that nondeterminism that is not present from the beginning, called *future nondeterministic choices* (FNCs) in Paper E, can generate an observer that is not correct if abstraction is performed before observer generation. In that paper, it is argued that abstractions of subsystems should not be performed before

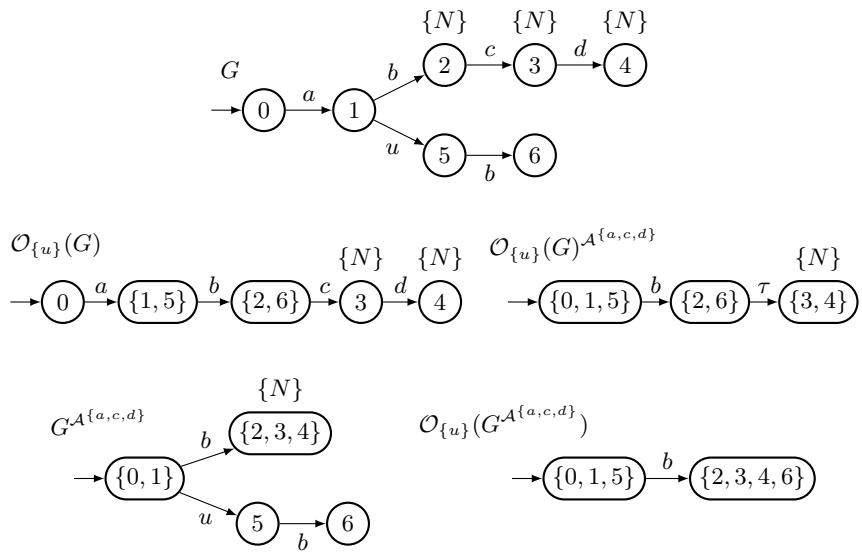

Figure 5.5: Observers with abstraction after and before observer generation. The correct abstracted observer is $\mathcal{O}_{\{u\}}(G)^{\mathcal{A}^{\{a,c,d\}}}$.

FNCs have been removed by synchronization of subsystems and observer generation. To be able to guarantee this, a formal definition of FNCs is given in the same paper. By that definition, it is possible to computationally decide if a system has any FNCs or not.

To summarize this subsection, before any abstractions can be performed, first any FNCs must have been removed by synchronization of subsystems and observer generation. Then, unique state labels must be added before and after unobservable events.

Algorithm

An algorithm for combined incremental observer generation and abstraction, in the presence of shared unobservable events, is obtained by a simple generalization of Algorithm 1 in Sect. 3.3. First, it is assumed that necessary local observers have been generated and synchronized, such that no FNCs exist anymore. The resulting observers are the input subsystems in Algorithm 1. Then, the only differences com-

pared to the original algorithm are that, before line 3, unique state labels for STSs of shared unobservable transitions are introduced in all submodels. Moreover, line 6 is replaced by

$$G_\Omega := \mathcal{O}((G_{\Omega_1})^{\mathcal{A}} \parallel (G_{\Omega_2})^{\mathcal{A}}).$$

In all abstractions, new local observable events are hidden, and in all observer generations, new local unobservable events are replaced by ε. After the observer generation on line 6, all obsolete state labels are also removed.

Finally, note that no observer generation is required when there are no shared unobservable events in G_{Ω_1} and G_{Ω_2}. A complement, in the heuristics on selection of the sets Ω_1 and Ω_2, is therefore to also focus on subsystems that have shared unobservable events as early as possible. In this way, extra observer generations can be significantly reduced.

5.4 Special Case: Local Unobservable Events

For a modular system, $G = \parallel_{i\in\mathbb{N}_n^+} G_i$, with partial observation, when all unobservable events are local, $\Sigma_{12}^{\varepsilon} = \varnothing$, it is proven in [52], [100], and more simply in Paper E, also including state labels, that an observer of the monolithic system $\mathcal{O}(G)$ can be computed by the synchronous composition of the local observers of its subsystems. Thus, (5.1) is simplified to

$$\mathcal{O}(G) = \parallel_{i\in\mathbb{N}_n^+} \mathcal{O}(G_i),$$

which also can be calculated by online synchronization as discussed in Chapter 2. Moreover, (5.2) results in

$$\mathcal{O}(G_1 \parallel G_2)^{\Sigma^h} \sim \left(\mathcal{O}(G_1)^{\mathcal{A}^{\Sigma_1^h}} \parallel O(G_2)^{\mathcal{A}^{\Sigma_2^h}}\right)^{\mathcal{A}^{\Sigma_{12}^h}}.$$

The algorithm for incremental abstraction is ones again Algorithm 1 in Sect. 3.3, where the input systems are now the local observers. After the generation of these observers, there can be some observer states that are *non-safe* and therefore considered to be *forbidden states*. The aim is to verify the existence of such forbidden states in a system, to conclude about its opacity/anonymity.

Since the problem only includes two types of states, safe and non-safe, these states properties can also be expressed in terms of marked and non-marked states. An alternative to generic state labels and visible bisimulation equivalence is then to transform

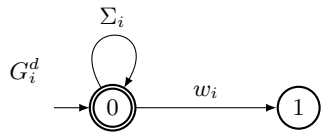

Figure 5.6: Detector automaton G_i^d that inserts blocking states, corresponding to the forbidden states.

the forbidden state problem to a *nonblocking problem*. Thus, conflict equivalence abstraction [6], [61], which preserves nonblocking, can be used.

Detector

Let all non-safe states in each individual observer $\mathcal{O}(G_i)$ be augmented with a self-loop. For CSO, these self-loops are labeled by w_i, $i = 1, \ldots, n$, and the resulting local observers are called $\mathcal{O}_{w_i}(G_i)$. For each such observer, a two-state *detector automaton* G_i^d, shown in Fig. 5.6, is then introduced. It includes a marked state with a self-loop on the set of observable events Σ_i in G_i and a transition via the event w_i to a non-marked state. The *extended local observer*

$$\mathcal{O}_e(G_i) = \mathcal{O}_{w_i}(G_i) \,\|\, G_i^d$$

then obtains non-marked blocking states added to every occurrence of a w_i self-loop in $\mathcal{O}_{w_i}(G_i)$. Thus, every forbidden state in $\mathcal{O}(G_i)$ results in a direct transition to a blocking state in the extended local observer, while all original states in $\mathcal{O}_{w_i}(G_i)$ become marked in $\mathcal{O}_e(G_i)$. If any blocking states remain in the total extended observer

$$\mathcal{O}_e(G) = \mathcal{O}_e(G_1) \,\|\, \mathcal{O}_e(G_2) \,\|\, \cdots \,\|\, \mathcal{O}_e(G_n),$$

this observer is blocking, and $\mathcal{O}(G)$ includes one or more non-safe states from a CSO point of view. In CSA case, the transformation is simplified by choosing the same self-loop label w for all observers $\mathcal{O}(G_i)$, $i = 1, \ldots, n$, and the same w label in every detector automaton G_i^d. This means that a blocking state may be reached first when all local observers have reached a non-safe state.

Example 6. Consider the subsystem G_i, $i \in \mathbb{N}_n^+$ in Fig. 5.7, where v_i is a local unobservable event. Moreover, c_i is a local observable event. The events a_i and a_{i+1} are observable but shared between neighbor subsystems, except for the local events

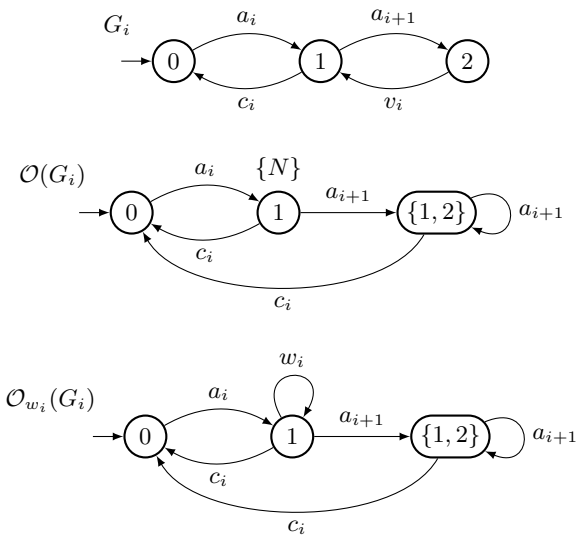

Figure 5.7: Transition system G_i, its local observer $\mathcal{O}(G_i)$ including state label N at the non-safe state, and local observer $\mathcal{O}_{w_i}(G_i)$ that is augmented with a w_i-self-loop at the non-safe state.

a_1 and a_{n+1}. The local observer $\mathcal{O}(G_i)$ is also shown in Fig. 5.7.

The transition system G_i is assumed to have one secret state, state 1. Thus, the observer state 1 is non-safe from a CSO point of view. This non-safe state with state

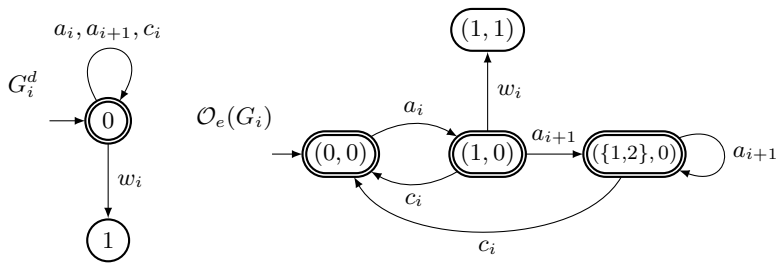

Figure 5.8: The detector automaton G_i^d that inserts an additional blocking state to the non-safe state in $\mathcal{O}_{w_i}(G_i)$, as depicted in $\mathcal{O}_e(G_i) = \mathcal{O}_{w_i}(G_i) \parallel G_i^d$.

label N in $\mathcal{O}(G_i)$ is a forbidden state to which a w_i self-loop is added in $\mathcal{O}_{w_i}(G_i)$ in Fig. 5.7. Including the detector G_i^d gives the extended local observer $\mathcal{O}_e(G_i) = \mathcal{O}_{w_i}(G_i) \parallel G_i^d$, as depicted in Fig. 5.8, where the w_i self-loop is replaced by a w_i transition to a blocking state. Utilizing the local observers, and combining them with incremental abstraction shows the strength of the approach in comparison to not using incremental abstraction, cf. Paper E. □

Enforcement of CSO and CSA

In order to determine whether a system satisfies a given specification or not, the system has to be *verified*, and if it fails, the system is restricted by *synthesizing* a *supervisor*. This means that states from which it is not possible to reach a desired marked state, are removed. Furthermore, any uncontrollable events that can be executed by the plant are not allowed to be disabled by the supervisor [101], [102]. Thus, a supervisor is synthesized to avoid blocking states and disabling uncontrollable events. Such a nonblocking and controllable supervisor is also maximally permissive, meaning that it restricts the system as little as possible.

In [103]–[106], it is shown how supervisory control can be adapted to enforce opacity, to disable some system behavior before they reveal a secret. In Paper E, it is shown for the first time how opacity and anonymity can be enforced by an observer based on maximally permissive supervisor that is generated by incremental abstraction. This supervisor generation follows naturally as an extension of the original forbidden state formulation of opacity and anonymity verification. The incremental abstraction is based on a supervision/synthesis equivalence, proposed in [107]–[109] as a natural extension of conflict equivalence [61].

CHAPTER 6

Summary of Appended Papers

This chapter provides a summary of the included papers.

6.1 Paper A

> **Mona Noori-Hosseini**, Bengt Lennartson, Maria Paola Cabasino, Carla Seatzu
> A survey on efficient diagnosability tests for automata and bounded Petri nets
> *18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2013, Cagliari, Italy.

In this paper, the efficiency of different polynomial diagnosability verifiers is evaluated for systems modeled by Petri nets. Since modular automata also can be represented by Petri nets, the method is applicable to both modeling formalisms.

Three verifiers based on three different polynomial algorithms are presented and compared to a fourth verifier, proposed in this paper. This verifier reduces the state space of the labeled system, by exploiting symmetry and abstracting unobservable transitions. All verifiers are generated based on either the reachability graph or the modified basis reachability graph (MBRG), a graph that is specifically proposed for diagnosability analysis of Petri nets. The importance of minimal explanations on the

performance of diagnosability verifiers is also shown. It is shown that the minimal explanation notion involved in the MBRG has great impact also on automata-based diagnosability methods.

The verifiers are compared in different aspects, and the results show that for small and medium sized systems or systems with significant number of unobservable transitions, our proposed verifier is more efficient. The evaluation shows improved computation times, often 1000 times faster or even more, when the most efficient automata-based diagnosability algorithms are combined with minimal explanations. Thus, significantly large systems can be analyzed when the most efficient automata-based diagnosability algorithms are combined with minimal explanations.

6.2 Paper B

Mona Noori-Hosseini and Bengt Lennartson
Diagnosability verification using compositional branching bisimulation
13th Workshop on Discrete Event Systems (WODES), 2016, Xi'an, China.

This paper presents an efficient diagnosability verification technique, based on a general abstraction approach. A general bisimilarity, called branching bisimulation including state labels and explicit divergence, is introduced based on a simple and efficient abstraction algorithm. Since this bisimulation preserves CTL*-X formulas, and diagnosability can be expressed as a CTL formula, this bisimulation is used in a compositional framework for modular diagnosability verification. It is shown that this compositional abstraction gives significant state space reduction, in comparison to state-of-the-art techniques.

The proposed method is general, and can be used to verify any CTL*-X formula for a set of synchronized subsystems, especially if the coupling between the subsystems only includes a few number of shared global events. This is illustrated by verifying non-diagnosability analytically for a set of synchronized subsystems, where the abstracted solution is independent of the number of subsystems and the number of observable events.

6.3 Paper C

Mona Noori-Hosseini, Bengt Lennartson
Incremental abstraction for diagnosability verification of modular systems
24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2019, Zaragoza, Spain.

This work presents an efficient approach to tackle the state space explosion problem, making it possible to verify diagnosability of large modular systems within a few seconds. In a diagnosability verifier with polynomial complexity, a non-diagnosable system generates uncertain loops. Such forbidden loops are transformed to forbidden states by simple automata-based detectors, which makes it possible to use efficient abstractions incrementally. The forbidden state problem is trivially transformed to a nonblocking problem by considering all states except the forbidden ones as marked states. This transformation is combined with one of the most efficient abstractions for modular systems called conflict equivalence, where nonblocking properties are preserved.

The incremental abstraction is applied to a scalable production system, including buffers and machines. For this modular system, the proposed diagnosability algorithm shows great results. Moreover, the computational time with and without incremental abstraction shows the great performance improvement of the proposed abstraction method.

6.4 Paper D

Mona Noori-Hosseini, Bengt Lennartson, Christoforos N. Hadjicostis
Compositional visible bisimulation abstraction applied to opacity verification
14th Workshop on Discrete Event Systems (WODES), 2018, Sorrento, Italy.

Visible bisimulation, proposed in this paper, is an equivalence-based definition of combined branching and stuttering bisimulation. It is shown how this bisimulation equivalence can be used to verify temporal logic expressions in an efficient way by compositional reduction, which is here called incremental abstraction. This bisimulation preserves all properties of a temporal logic called ECTL*, where CTL* is extended with events.

The presented bisimulation abstraction is applied to a set of synchronized subsystems, where local events are identified incrementally and abstracted after each syn-

chronization. Since the bisimulation reduction is applied after each synchronization, a significant part of the state space explosion in ordinary synchronization is avoided. This incremental abstraction is used for opacity verification, where it is shown that local observers can be generated before they are synchronized. This is a key factor to be able to apply incremental abstraction to opacity verification for large modular system. The efficiency of this method is illustrated on a modular opacity problem with mutual exclusion of moving agents. The results show a great potential to solve opacity problems for large modular systems.

6.5 Paper E

Mona Noori-Hosseini, Bengt Lennartson, Christoforos N. Hadjicostis
Incremental observer abstraction for opacity/privacy verification and enforcement
Submitted for possible journal publication. An invitation to submit a revised version has been received, 2019.

Verification of two security/privacy notions, current state opacity and anonymity, is considered in this paper. An incremental observer generation for modular systems is presented, where the computational complexity is alleviated by local observer generation and incremental abstraction. For the case that shared unobservable events are also involved, a new combined incremental abstraction and observer generation is proposed. This requires some precautions to be able to accomplish local abstractions before shared unobservable events are removed by observer generation. Temporary state labels are then added to achieve necessary restrictions.

The combined incremental abstraction and observer generation requires an abstraction that includes state labels and is congruent with respect to synchronization and hiding. In this paper, visible bisimulation is used for that purpose. It is also shown how current state opacity and anonymity can be enforced by a supervisor. This is achieved by a natural extension of the verification problem to a supervisory control problem based on forbidden states and incremental abstraction. Finally, a modular and scalable building security problem with arbitrary number of floors and elevators is presented, for which the efficiency of the proposed incremental verification and synthesis procedures is demonstrated.

CHAPTER 7

Concluding Remarks and Future Research

The main goal of this thesis is to achieve efficient formal verification of safety and security/privacy properties in modular discrete event systems. The main issue in formal verification of large and complex modular systems is the state space explosion, which can be mitigated using abstraction methods. Different abstraction methods are presented. Moreover, incremental abstraction, as a technique that achieves step-by-step reduction of modular systems, is applied. This method adds more reduction, in comparison to the methods that only abstract once. In this thesis, a general abstraction technique, called *visible bisimulation abstraction*, is proposed. The purpose is to reduce transition systems that include both state and event labels. Visible bisimulation preserves temporal logic and divergence properties when incremental abstraction is applied.

This powerful incremental abstraction is used for fault diagnosability verification of modular discrete event systems of industrial size. Some necessary considerations are developed, in order to be able to apply incremental abstraction for diagnosability verification using verifier techniques. Furthermore, it is shown that the diagnosability problem can be transformed to a nonblocking verification problem using a three-state detector. This implies that a very efficient incremental abstraction that preserves the nonblocking property can be applied.

It is also shown that different diagnosability verifiers can be applied for diagnosability verification of Petri nets, when the Petri net model is transformed to an automaton by generating a modified basis reachability graph. This graph is generated by performing abstractions on the Petri net, which generates a more compact representation of the original model.

For current state opacity and anonymity verification of modular systems, an incremental approach is utilized as well, which shows great improvements when the previously mentioned abstraction methods are applied. As a nontrivial extension, the existence of shared unobservable events in modular systems is also considered, where the incremental abstraction is generalized to a alternating incremental abstraction and observer generation. In the analysis of this alternating procedure, it is shown that some precautions are required to be able to accomplish local abstractions before shared unobservable events are removed by observer generation. It is also shown how current state opacity and anonymity can be enforced by a supervisor, that is generated by identifying forbidden states and ones again applying incremental abstraction. For all problems considered in this thesis, it is demonstrated that different proposed incremental abstractions result in efficient state space reductions, enabling us to formally verify and synthesize large systems in a short time.

7.1 Future Research

Some possible extensions of this work are also observed:

- The proposed incremental verification of temporal logic formulas should be compared with available model checking algorithms. Some examples show a great potential for our method, applied to large modular systems.
- The incremental abstraction approach can be applied to temporal logic verification of Petri nets, and diagnosability verification of models with different structures.
- Opacity verification can be extended to more general settings, for instance having more than one moving object in a system.
- For synthesis purpose, visible bisimulation abstraction can be extended to include uncontrollable events.
- The combined incremental abstraction and observer generation can also be applied to opacity and anonymity supervisor enforcement, for modular systems

including shared unobservable events.

References

[1] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2008.

[2] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Second. Addison-Wesley, 2001, ISBN: 0201441241.

[3] M. P. Cabasino, A. Giua, and C. Seatzu, "Introduction to Petri nets", in *Lecture Notes in Control and Information Sciences*, Springer, 2013, pp. 191–211.

[4] C. Baier and J. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[5] S. Graf, B. Steffen, and G. Lüttgen, "Compositional minimisation of finite state systems using interface specifications", *Formal Aspects of Computing*, vol. 8, no. 5, pp. 607–616, 1996.

[6] H. Flordal and R. Malik, "Compositional verification in supervisory control", *SIAM Journal on Control and Optimization*, vol. 48, no. 3, pp. 1914–1938, 2009.

[7] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis, "Diagnosability of discrete-event systems", *IEEE Transactions on Automatic Control*, vol. 40, no. 9, pp. 1555–1575, 1995.

[8] S. Jiang, Z. Huang, V. Chandra, and R. Kumar, "A polynomial algorithm for testing diagnosability of discrete-event systems", *IEEE Transactions on Automatic Control*, vol. 46, no. 8, pp. 1318–1321, 2001.

[9] K. Schmidt, "Abstraction-based failure diagnosis for discrete event systems", *Systems & Control Letters*, vol. 59, no. 1, pp. 42–47, 2010.

[10] T.-S. Yoo and S. Lafortune, "Polynomial-time verification of diagnosability of partially observed discrete-event systems", *IEEE Transactions on Automatic Control*, vol. 47, no. 9, pp. 1491–1495, 2002.

[11] M. Moreira, T. Jesus, and J. Basilio, "Polynomial time verification of decentralized diagnosability of discrete event systems", *IEEE Transactions on Automatic Control*, vol. 56, no. 7, pp. 1679–1684, 2011.

[12] O. Contant, S. Lafortune, and D. Teneketzis, "Diagnosability of discrete event systems with modular structure", *Discrete Event Dynamic Systems*, vol. 16, no. 1, pp. 9–37, 2006.

[13] D. Myadzelets and A. Paoli, "Virtual modules in discrete event systems: Achieving modular diagnosability", Tech. Rep. https://arxiv.org/abs/1311.2850, 2013.

[14] B. Li, J. C. Basilio, M. Khlif-Bouassida, and A. Toguyéni, "Polynomial time verification of modular diagnosability of discrete event systems", *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 13 618–13 623, 2017.

[15] K. W. Schmidt, "Verification of modular diagnosability with local specifications for discrete-event systems", *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 5, pp. 1130–1140, 2013.

[16] M. P. Cabasino, A. Giua, and C. Seatzu, "Diagnosability of discrete-event systems using labeled Petri nets", *IEEE Transactions on Automation Science and Engineering*, vol. 11, no. 1, pp. 144–153, 2014.

[17] R. Focardi and R. Gorrieri, "A taxonomy of trace-based security properties for CCS", in *Proceedings The Computer Security Foundations Workshop VII*, IEEE Comput. Soc. Press, 1996, pp. 126–136.

[18] A. Saboori and C. N. Hadjicostis, "Notions of security and opacity in discrete event systems", in *46th IEEE Conference on Decision and Control*, 2007, pp. 5056–5061.

[19] R. Jacob, J.-J. Lesage, and J.-M. Faure, "Overview of discrete event systems opacity: Models, validation, and quantification", *Annual Reviews in Control*, vol. 41, pp. 135–146, 2016.

[20] J. W. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan, "Opacity generalised to transition systems", *International Journal of Information Security*, vol. 7, no. 6, pp. 421–435, 2008.

[21] A. Saboori and C. N. Hadjicostis, "Opacity-enforcing supervisory strategies for secure discrete event systems", in *47th IEEE Conference on Decision and Control*, 2008.

[22] ——, "Current-state opacity formulations in probabilistic finite automata", *IEEE Transactions on Automatic Control*, vol. 59, no. 1, pp. 120–133, 2014.

[23] J. W. Bryans, M. Koutny, and P. Y. Ryan, "Modelling opacity using Petri nets", *Electronic Notes in Theoretical Computer Science*, vol. 121, pp. 101–115, 2005.

[24] Y. Tong, Z. Ma, Z. Li, C. Seactzu, and A. Giua, "Verification of language-based opacity in Petri nets using verifier", in *American Control Conference (ACC)*, 2016.

[25] Y. Tong, Z. Li, C. Seatzu, and A. Giua, "Verification of state-based opacity using Petri nets", *IEEE Transactions on Automatic Control*, vol. 62, no. 6, pp. 2823–2837, 2017.

[26] S. Lafortune, F. Lin, and C. N. Hadjicostis, "On the history of diagnosability and opacity in discrete event systems", *Annual Reviews in Control*, vol. 45, pp. 257–266, 2018.

[27] A. Saboori and C. N. Hadjicostis, "Coverage analysis of mobile agent trajectory via state-based opacity formulations", *Control Engineering Practice*, vol. 19, no. 9, pp. 967–977, 2011.

[28] E. Badouel, M. Bednarczyk, A. Borzyszkowski, B. Caillaud, and P. Darondeau, "Concurrent secrets", in *8th International Workshop on Discrete Event Systems*, 2006.

[29] F. Cassez, "The dark side of timed opacity", in *Advances in Information Security and Assurance*, Springer, 2009, pp. 21–30.

[30] F. Lin, "Opacity of discrete event systems and its applications", *Automatica*, vol. 47, no. 3, pp. 496–503, 2011.

[31] Y.-C. Wu and S. Lafortune, "Comparative analysis of related notions of opacity in centralized and coordinated architectures", *Discrete Event Dynamic Systems*, vol. 23, no. 3, pp. 307–339, 2013.

[32] Y.-C. Wu, K. A. Sankararaman, and S. Lafortune, "Ensuring privacy in location-based services: An approach based on opacity enforcement", *IFAC Proceedings Volumes*, vol. 47, no. 2, pp. 33–38, 2014.

[33] A. Saboori and C. N. Hadjicostis, "Verification of k-step opacity and analysis of its complexity", *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 3, pp. 549–559, 2011.

[34] ——, "Verification of initial-state opacity in security applications of discrete event systems", *Information Sciences*, vol. 246, pp. 115–132, 2013.

[35] B. Wu and H. Lin, "Privacy verification and enforcement via belief abstraction", *IEEE Control Systems Letters*, vol. 2, no. 4, pp. 815–820, 2018.

[36] S. Mohajerani and S. Lafortune, "Transforming opacity verification to nonblocking verification in modular systems", *IEEE Transactions on Automatic Control, in press*, 2020.

[37] A. Bourouis, K. Klai, N. B. Hadj-Alouane, and Y. E. Touati, "On the verification of opacity in web services and their composition", *IEEE Transactions on Services Computing*, vol. 10, no. 1, pp. 66–79, 2017.

[38] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams", *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[39] K. Zhang and M. Zamani, "Infinite-step opacity of nondeterministic finite transition systems: A bisimulation relation approach", in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 2017.

[40] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[41] R. J. V. Glabbeek and W. P. Weijland, "Branching time and abstraction in bisimulation semantics", *Journal of the ACM*, vol. 43, pp. 555–600, 1996.

[42] M. Noori-Hosseini, B. Lennartson, M. P. Cabasino, and C. Seatzu, "A survey on efficient diagnosability tests for automata and bounded Petri nets", in *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2013.

[43] M. Noori-Hosseini and B. Lennartson, "Diagnosability verification using compositional branching bisimulation", in *13th International Workshop on Discrete Event Systems (WODES)*, 2016.

[44] M. Noori-Hosseini, B. Lennartson, and C. Hadjicostis, "Compositional visible bisimulation abstraction applied to opacity verification", *IFAC: 14th International Workshop on Discrete Event Systems (WODES)*, vol. 51, no. 7, pp. 434–441, 2018.

[45] M. Noori-Hosseini, B. Lennartson, and C. N. Hadjicostis, "Incremental observer reduction applied to opacity verification and synthesis", Tech. Rep. https://arxiv.org/abs/1812.08083, 2019.

[46] M. Noori-Hosseini and B. Lennartson, "Incremental abstraction for diagnosability verification of modular systems", *24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2019)*, 2019.

[47] B. Lennartson and M. Noori-Hosseini, "Visible bisimulation equivalence — a unified abstraction for temporal logic verification", *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 400–407, 2018.

[48] C. Belta, B. Yordanov, and E. A. Gol, "Formal methods for discrete-time dynamical systems, studies in systems, decision and control", *Springer*, vol. 89, 2017.

[49] R. Nicola and F. Vaandrager, "Action versus state based logics for transition systems", in *Semantics of Systems of Concurrent Processes*, Springer, 1990, pp. 407–419.

[50] C. Hoare, "Communicating sequential processes", *Science of Computer Programming, Prentice-Hall International, London*, vol. 9, no. 1, pp. 101–105, 1987.

[51] M. Cabasino, A. Giua, and C. Seatzu, "Diagnosability of bounded Petri nets", *48th IEEE Conference on Decision and Control (CDC)*, pp. 1254–1260, 2009.

[52] E. Fabre, "Diagnosis and automata", in *Control of Discrete-Event Systems: Automata and Petri net Perspectives*, ser. Lecture Notes in Control and Information Sciences, C. Seatzu, M. Silva, and J. H. van Schuppen, Eds., Springer, 2012, pp. 85–106.

[53] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking", in *Computer Aided Verification*, Springer, 2002, pp. 359–364.

[54] M. Huth and M. Ryan, *Logic in Computer Science, modelling and reasoning about systems*. Cambridge University Press, 2009.

[55] R. D. Nicola and F. Vaandrager, "Three logics for branching bisimulation", *Journal of the ACM*, vol. 42, pp. 458–487, 1995.

[56] R. Gerth, R. Kuiper, D. Peled, and W. Penczek, "A partial order approach to branching time logic model checking", *Journal of the ACM*, vol. 150, pp. 132–152, 1999.

[57] N. Trčka, *Silent Steps in Transition Systems and Markov Chains*, IPA Dissertation Series 2007-08, Technische Universiteit Eindhoven, 2007, 1996.

[58] R. J. Van Glabbeek, B. Luttik, and N. Trčka, "Branching bisimilarity with explicit divergence. the semantics of sequential systems with silent moves", *Fundamenta Informaticae*, vol. 93, pp. 371–392, 2009.

[59] S. Blom and S. Orzan, "Distributed branching bisimulation reduction of state spaces", *Electronic Notes in Theoretical Computer Science*, vol. 89, pp. 99–113, 2003.

[60] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications", *Pacific Journal of Mathematics*, vol. 5, pp. 285–309, 1955.

[61] R. Malik, D. Streader, and S. Reeves, "Fair testing revisited: A process-algebraic characterisation of conflicts", in *Automated Technology for Verification and Analysis*, Springer, 2004, pp. 120–134.

[62] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, "Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems", in *8th international Workshop on Discrete Event Systems, WODES'06*, 2006, pp. 384–385.

[63] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker", in, ser. Lecture Notes in Computer Science, vol. 8559, Springer, 2014, pp. 334–342.

[64] K. Claessen and N. Sorensson, "A liveness checking algorithm that counts", in *Formal Methods in Computer-Aided Design (FMCAD)*, G. Cabodi and S. Singh, Eds., IEEE, 2012, pp. 52–59.

[65] B. Parhami, "Defect, fault, error,..., or failure?", *IEEE Transactions on Reliability*, vol. 46, no. 4, pp. 450–451, 1997.

[66] B. Randell, "On failures and faults", in *FME 2003: Formal Methods*, Springer Berlin Heidelberg, 2003, pp. 18–39.

[67] F. Lin, "Diagnosability of discrete event systems and its applications", *Discrete Event Dynamic Systems: Theory and Applications*, vol. 4, no. 2, pp. 197–212, 1994.

[68] Z. Huang, S. Bhattacharyya, R. Kumar, S. Jiang, and V. Chandra, "Diagnosis of discrete-event systems in rules-based model using first-order linear temporal logic", *Asian Journal of Control*, pp. 1–9, 2008.

[69] W. Qiu and R. Kumar, "Decentralized failure diagnosis of discrete event systems", *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 36, no. 2, pp. 384–395, 2006.

[70] F. Basile, P. Chiacchio, and G. D. Tommasi, "An efficient approach for online diagnosis of discrete event systems", *IEEE Transactions on Automatic Control*, vol. 54, no. 4, pp. 748–759, 2009.

[71] F. G. Cabral, M. V. Moreira, and O. Diene, "Online fault diagnosis of modular discrete-event systems", in *54th IEEE Conference on Decision and Control (CDC)*, 2015.

[72] M. Cabasino, A. Giua, M. Pocci, and C. Seatzu, "Discrete event diagnosis using labeled Petri nets. an application to manufacturing systems", *Control Engineering Practice*, vol. 19, pp. 989–1001, 2011.

[73] C. Zhou, R. Kumar, and R. S. Sreenivas, "Decentralized modular diagnosis of concurrent discrete event systems", in *9th International Workshop on Discrete Event Systems*, 2008.

[74] T.-S. Yoo and H. E. Garcia, "Diagnosis of behaviors of interest in partially-observed discrete-event systems", *Systems & Control Letters*, vol. 57, no. 12, pp. 1023–1029, 2008.

[75] S. Zad, R. Kwong, and W. Wonham, "Fault diagnosis in discrete-event systems: Framework and model reduction", *IEEE Transactions on Automatic Control*, vol. 48, no. 7, pp. 1199–1212, 2003.

[76] Y. Pencolé and M.-O. Cordier, "A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks", *Artificial Intelligence*, vol. 164, no. 1-2, pp. 121–170, 2005.

[77] R. Debouk, S. Lafortune, and D. Teneketzis, "Coordinated decentralized protocols for failure diagnosis of discrete event systems", *Discrete Event Dynamic Systems*, vol. 10, no. 1, pp. 33–86, 2000.

[78] K. Schmidt, "Abstraction-based verification of codiagnosability for discrete event systems", *Automatica*, vol. 46, no. 9, pp. 1489–1494, 2010.

[79] Y. Wang, T.-S. Yoo, and S. Lafortune, "Erratum to: Diagnosis of discrete event systems using decentralized architectures", *Discrete Event Dynamic Systems*, vol. 25, no. 4, pp. 601–603, 2013.

[80] N. Ran, H. Su, A. Giua, and C. Seatzu, "Codiagnosability analysis of bounded Petri nets", *IEEE Transactions on Automatic Control*, vol. 63, no. 4, pp. 1192–1199, 2018.

[81] S. L. Ricker and J. H. van Schuppen, "Decentralized failure diagnosis with asynchronous communication between supervisors", in *European Control Conference (ECC)*, 2001.

[82] W. Qiu and R. Kumar, "Distributed diagnosis under bounded-delay communication of immediately forwarded local observations", *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 38, no. 3, pp. 628–643, 2008.

[83] H. Khorasgani, D. Jung, and G. Biswas, "Structural approach for distributed fault detection and isolation", *IFAC-PapersOnLine*, vol. 48, no. 21, pp. 72–77, 2015.

[84] C. G. P. Zuniga, "Structural analysis for the diagnosis of distributed systems", PhD thesis, Automatic. INSA de Toulouse, 2018.

[85] R. Debouk, R. Malik, and B. Brandin, "A modular architecture for diagnosis of discrete event systems", in *Proceedings of the 41st Conference on Decision and Control (CDC), 2002*.

[86] T.-S. Yoo and S. Lafortune, "Polynomial-time verification of diagnosability of partially observed discrete-event systems", *IEEE Transactions on Automatic Control*, vol. 47, no. 9, pp. 1491–1495, 2002.

[87] M. Cabasino, A. Giua, and C. Seatzu, "Fault detection for discrete event systems using Petri nets with unobservable transitions", *Automatica*, vol. 46, no. 9, pp. 1491–1495, 2010.

[88] 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)*, 2003, pp. 363–369.

[89] A. Boussif and M. Ghazel, "Intermittent fault diagnosis of industrial systems in a model-checking framework", in *IEEE International Conference on Prognostics and Health Management (ICPHM)*, 2016, pp. 1–6.

[90] S. Jiang and R. Kumar, "Failure diagnosis of discrete-event systems with linear-time temporal logic specifications", *IEEE Transactions on Automatic Control*, vol. 49, no. 6, pp. 934–945, 2004.

[91] S. Jiang and R. Kumar, "Diagnosis of repeated failures for discrete event systems with linear-time temporal-logic specifications", *IEEE Transactions on Automation Science and Engineering*, vol. 3, no. 1, pp. 47–59, 2006.

[92] J. Chen and R. Kumar, "Fault detection of discrete-time stochastic systems subject to temporal logic correctness requirements", *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 4, pp. 1369–1379, 2015.

[93] A. Giua, S. Lafortune, and C. Seatzu, "Divergence properties of labeled Petri nets and their relevance for diagnosability analysis", *IEEE Transactions on Automatic Control, in press*, 2019.

[94] M. Noori-Hosseini and B. Lennartson, "Verification of diagnosability based on compositional branching bisimulation", in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014.

[95] Y. Ru and C. N. Hadjicostis, "Fault diagnosis in discrete event systems modeled by partially observed Petri nets", *Discrete Event Dynamic Systems*, vol. 19, no. 4, pp. 551–575, 2009.

[96] G. Jiroveanu and R. Boel, "The diagnosability of Petri net models using minimal explanations", *IEEE Transactions on Automatic Control*, vol. 55, no. 7, pp. 1663–1668, 2010.

[97] M. P. Cabasino, A. Giua, and C. Seatzu, "Fault detection for discrete event systems using Petri nets with unobservable transitions", *Automatica*, vol. 46, no. 9, pp. 1531–1539, 2010.

[98] S. Lafortune and F. Lin, "From diagnosability to opacity: A brief history of diagnosability or lack thereof", *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 3022–3027, 2017.

[99] T. Masopust, “Critical observability for automata and Petri nets”, Tech. Rep. https://arxiv.org/abs/1808.00261, 2018.

[100] G. Pola, E. D. Santis, M. D. D. Benedetto, and D. Pezzuti, “Design of decentralized critical observers for networks of finite state machines: A formal method approach”, *Automatica*, vol. 86, pp. 174–182, 2017.

[101] P. Ramadge and W. Wonham, “The control of discrete event systems”, *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[102] W. Wonham, K. Cai, and K. Rudie, “Supervisory control of discrete-event systems: A brief history – 1980-2015”, *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 1791–1797, 2017.

[103] Y. Falcone and H. Marchand, “Enforcement and validation (at runtime) of various notions of opacity”, *Discrete Event Dynamic Systems*, vol. 25, no. 4, pp. 531–570, 2015.

[104] J. Dubreil, P. Darondeau, and H. Marchand, “Supervisory control for opacity”, *IEEE Transactions on Automatic Control*, vol. 55, no. 5, pp. 1089–1100, 2010.

[105] A. Saboori and C. N. Hadjicostis, “Opacity-enforcing supervisory strategies via state estimator constructions”, *IEEE Transactions on Automatic Control*, vol. 57, no. 5, pp. 1155–1165, 2012.

[106] S. Mohajerani, Y. Ji, and S. Lafortune, “Compositional and abstraction-based approach for synthesis of edit functions for opacity enforcement”, *IEEE Transactions on Automatic Control, in press*, 2020.

[107] H. Flordal, R. Malik, M. Fabian, and K. Åkesson, “Compositional Synthesis of Maximally Permissive Supervisors Using Supervision Equivalence”, *Discrete Event Dynamic Systems*, vol. 17, no. 4, pp. 475–504, 2007.

[108] S. Mohajerani, R. Malik, and M. Fabian, “A framework for compositional synthesis of modular nonblocking supervisors”, *IEEE Transactions on Automatic Control*, vol. 59, no. 1, pp. 150–162, 2014.

[109] ——, “Compositional synthesis of supervisors in the form of state machines and state maps”, *Automatica*, vol. 76, pp. 277–281, 2017.