UNIVERSITÀ DEGLI STUDI DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

Formal Failure Analyses for Effective Fault Management
An Aerospace Perspective

Benjamin Bittner

Advisor
Dr. Alessandro Cimatti
Fondazione Bruno Kessler

Co-Advisor
Dr. Marco Bozzano
Fondazione Bruno Kessler

December 2016

Abstract

The possibility of failures is a reality that all modern complex engineering systems need to deal with. In this dissertation we consider two techniques to analyze the nature and impact of faults on system dynamics, which is fundamental to reliably manage them.

Timed failure propagation analysis studies how and how fast faults propagate through physical and logical parts of a system. We develop formal techniques to validate and automatically generate representations of such behavior from a more detailed model of the system under analysis.

Diagnosability analysis studies the impact of faults on observable parameters and tries to understand whether the presence of faults can be inferred from the observations within a useful time frame. We extend a recently developed framework for specifying diagnosis requirements, develop efficient algorithms to assess diagnosability under a fixed set of observables, and propose an automated technique to select optimal subsets of observables.

The techniques have been implemented and evaluated on realistic models and case studies developed in collaboration with engineers from the European Space Agency, demonstrating the practicality of the contributions.

Keywords

[Failure Analysis, Timed Failure Propagation, Diagnosability, Symbolic Model-Checking, Model Based Engineering]

Acknowledgements

I would like to thank my supervisors Alessandro Cimatti and Marco Bozzano for guiding and supporting me in these four years of research and in writing this dissertation. I gratefully acknowledge the support received from the European Space Agency through the NPI program, which gave me the opportunity to apply my research to interesting industrial problems. Many thanks to Marcel, Andrei, and Yuri at ESA, who were always available for answering my many questions. Many thanks go to all my colleagues at Fondazione Bruno Kessler. Thanks to Marco and Andrea for reviewing parts of the dissertation. Finally I would like to express my gratitude to my wife that supported me with her love and understanding throughout this journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern complex engineering systems, such as satellites, airplanes and traffic control systems need to be able to handle faults. In general, faults can be seen as states in which particular components or larger parts of a system are no longer able to perform their required function. Faults can compromise system safety, creating a risk of damage to the system itself, damage to surrounding property or infrastructure, both virtual or physical, or even a risk of harm to humans. Beyond creating some sort of damage, faults can also affect the availability of a system, for instance by causing service outages. In some applications such as telecom satellites or intelligence infrastructure, such outages might be unacceptable. As a consequence, when faults are present, the system needs to either tolerate them passively, for instance through robust control laws, or manage them actively in order to guarantee system operations according to the requirements.

A widely used approach to implement fault management is FDIR, short for *Fault Detection, Isolation, and Recovery.* The European Cooperation for Space Standardization describes FDIR as follows:

> *The overall on-board fault management concept is based on the failure detection, isolation and recovery (FDIR) paradigm. This means that functions are implemented:*

- *to detect on-board failures and to report them to the relevant on-board units or subsystems and to the ground segment;*
- *to isolate the failure, i.e. to avoid the propagation of the failure and the deterioration of the equipment;*
- [...] *to recover the on-board functions affected by the failure such that mission operations can continue.*

(ECSS-E-ST-70-11C, Section 5.7.5, On-board fault management)

The data and control flow in FDIR is typically structured as shown in Figure 1.1. The FDIR observes the system using hardware sensor readings or values computed by the software. When a fault appears, the detection component (ideally) recognizes this by reasoning on the observations and raises an alarm. The reasoning can be done by a simple monitor that triggers when a Boolean condition over the observations becomes true, or by a more complex diagnoser that reasons on the evolution of the observations over time. Once an alarm is being raised, the FDIR will issue commands to the system to stop the fault from propagating to other components or logical system partitions (isolation) such that safety is reestablished, and possibly to take the system back to a normal operating state (recovery).

In this context the present dissertation explores formal techniques to analyze the impact of faults on a system. A precise understanding of how faults affect a system is fundamental for developing effective and efficient fault management solutions. We generally adopt an aerospace perspective when describing architectural assumptions and possible applications of the contributions, as much of the work included here was motivated by industrial projects funded by the European Space Agency (ESA). This does not mean that the core results of the dissertation are only applicable to aerospace engineering problems: in the related literature many other application examples are mentioned, from complex computer systems to power

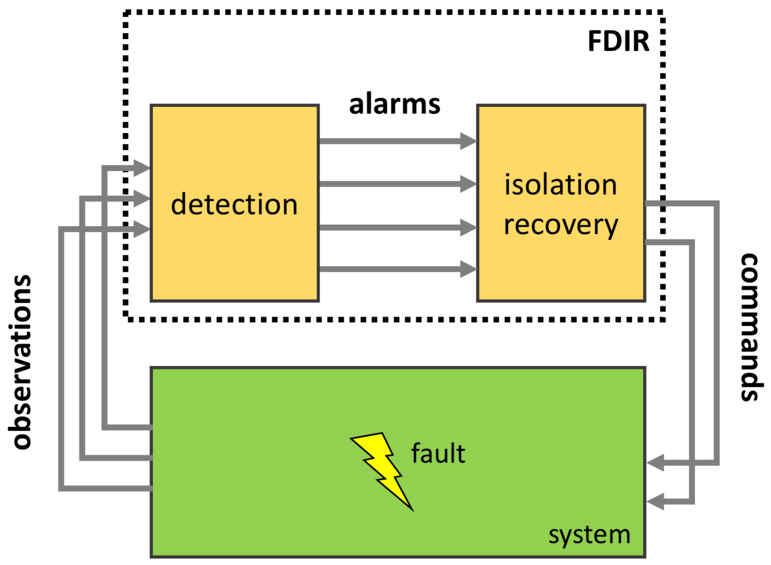


Figure 1.1: Common FDIR flow.

distribution plants and heating, ventilation, and air conditioning systems.

Before an FDIR design is developed, three questions need to be answered to understand the feasibility of this task:

Propagation How do faults propagate through various physical and logical parts of the system in terms of deviations from nominal state, and how fast can these propagations be?

Diagnosability Will these propagations affect the observations available to FDIR in a way that allows detection and identification of the faults in time to trigger a recovery?

Recoverability Are the control means sufficient to stop the propagation, once detected, and return the system to a normal operating state?

First, it is important to describe the propagation effects that the FDIR has to deal with, and how much time it has to react before effects of a higher severity level occur. This type of analysis aims at supporting the identification of a complete set of requirements on FDIR. For propagation analysis we work with timed failure propagation graphs (TFPG). These models allow a more comprehensive and integrated description of failure propagation compared to classical techniques, and include also information on propagation speed. We develop a framework and corresponding algorithms to validate TFPGs against a more detailed representation of the system's behavior, as well as a method to automatically compute them from such system models. The developed techniques have been evaluated on industrial case studies from the aerospace domain. These have in part been developed at the European Space Agency technology centre ESTEC, based on a space mission currently under development. The evaluation shows the feasibility and practicality of the developed analysis techniques.

The results of propagation analysis are then used to identify the conditions that need to be diagnosed, and the time bounds within which diagnosis has to occur. Diagnosability analysis checks whether these conditions can be inferred within the given time bound, based on the information provided by observations. Our contributions in this area are based on a recently developed framework for specification of expressive diagnosis requirements. We develop efficient algorithms to test if diagnosis can actually be performed under a given set of observables, as well as algorithms for identifying subsets of the observables that still guarantee diagnosability and possibly optimize a cost function. Also for these contributions benchmarks on industrial models were performed, demonstrating the scalability of the techniques. The models were derived from collaborations with NASA and Boeing.

At this point we assume to have a set of alarms that detect faulty conditions of interest. The next step is to check whether, as soon as these alarms trigger, the available control means can be used to put the system into a safe state and possibly also resume operations. TFPGs can support recoverability analysis by deriving requirements on how fast the recovery needs to be executed to prevent further damage, and what the propagation behaviors are that need to be stopped. A more thorough assessment of recoverability would require considering also discrete control theory, but this is out of scope for the present dissertation.

Thus, we want to support derivation of requirements on what the FDIR has to achieve, and to check whether it is feasible to implement an FDIR that can satisfy those requirements. The task of actually building the FDIR is a distinct problem which is out of scope for the present work.

The dissertation focuses on systems that can be described mostly in terms of discrete states and events, as opposed to physical modeling that focuses on the continuous evolution of physical quantities over time. This discrete view is commonly adopted when dealing with system-level control, where, conceptually, most properties and dynamics can be described in discrete terms [Cassandras and Lafortune, 2008], and has been adopted also for FDIR analysis in industrial case studies – see e.g. [Pintard et al., 2012, Rugina et al., 2012].

Thesis Structure and Contributions

We now describe the structure of the dissertation and summarize the contributions.

In Chapter 2 we introduce the relevant background on which the following chapters build. It describes a framework to model dynamic systems, two temporal logics to express properties on the behaviors of such systems, and a formal verification technique (model-checking) to check whether these properties hold for all possible system behaviors.

In Chapter 3 we describe the following contributions for timed failure propagation analysis:

1. Trace-based semantics for TFPGs are developed to formally treat

them as an abstraction of dynamic system models.

2. Formal properties are defined describing how well TFPGs abstract a system model of reference.

3. The problem of checking those properties is mapped to a model-checking problem.

4. An algorithm is developed to automatically generate TFPGs with well-defined characteristics from a given system model.

The developed algorithms were implemented and tested on representative models, including some derived from industrial projects. The results on TFPGs were published in [Bittner et al., 2016c] and [Bittner et al., 2016b]. The integration of the developed software in a toolkit for safety analysis is described in [Bittner et al., 2016a].

In Chapter 4, building on a recently developed framework for the specification and generation of on-line model-based diagnosers, the following contributions to diagnosability analysis are described:

1. The framework is extended to include specification of an operational context within which diagnosability must be guaranteed.

2. The classical way to falsify diagnosability is shown to represent a necessary and sufficient condition in many important cases, but due to the increased framework expressiveness only a necessary condition in corner cases.

3. The problem of verifying diagnosability is reduced to a model-checking problem.

4. The problem of identifying subsets of observables that optimize a cost function is reduced to a parametric version of model-checking.

The algorithms are implemented and evaluated in an extensive experimental evaluation. Partial results were published in [Bittner et al., 2012] and [Bittner et al., 2014a], and a journal article summarizing the chapter is being prepared for submission.

In Chapter 5 the industrial application of the developed techniques is studied, resulting in the following contributions:

- A method to translate TFPGs into state machines is described to support diagnoser synthesis.
- Two case studies on TFPG modeling for an aerospace project demonstrate the adequacy of the framework and implementation.
- A third case study on the same project evaluates the application of TFPGs to FDIR review, FDIR tuning, and diagnostic support.

The integration of TFPGs in a novel FDIR development process is described in [Bittner et al., 2014b].

Finally, Chapter 6 concludes the thesis by summarizing the results and providing an outlook on future work.

Chapter 2

Technical Background

We now describe the formal framework used in this dissertation. First a set of basic logical notions are introduced in Section 2.1. Then, in Section 2.2, we define symbolic transition systems as the formalism of choice for encoding the behavior of dynamic systems. To characterize temporal behavior of such systems we will use temporal logic, defined in Section 2.3. Finally we introduce the problem of model-checking of such properties on a transition system in Section 2.4. This formal background will be used in the subsequent chapters to study timed failure propagation and diagnosability in dynamic systems.

2.1 Preliminaries

We work in the setting of predicate logic [van Dalen, 1994]. Terms are either constants, individual variables, or the application of an n-ary function symbol to n terms. Atomic propositions are either Boolean variables, equality between two terms, or the application of an n-ary predicate symbol to n terms. A formula is either an atomic proposition or the application of a Boolean connective (negation $\neg$, conjunction $\wedge$, disjunction $\vee$) to formulae. We use the following abbreviations: $\phi \wedge \psi$ for $\neg(\neg\phi \vee \neg\psi)$, $\phi \rightarrow \psi$ for $\neg\phi \vee \psi$, and $\phi \leftrightarrow \psi$ for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. With a slight abuse of notation we may sometimes write $P = Q$ instead of $P \leftrightarrow Q$ when P and Q are propositional variables.

We use the standard notions of assignment, model, and logical consequence. The domain of a variable x is represented as $\Delta(x)$. Given an assignment μ to a set of variables X, and $X_1 \subseteq X$, we denote the projection of μ over X_1 with $Proj(\mu, X_1)$. If X is a set of variables and $\Box$ is a symbol, then $X_\Box$ stands for the set of variables $\{x_\Box \mid x \in X\}$. We write $\phi(X)$ to stress that ϕ is a formula over the variables in X. $\phi(X_\Box)$ is the formula obtained from ϕ by the parallel substitution of every occurrence of x with $x_\Box$, for all $x \in X$. A formula ϕ can be simplified with respect to a (partial) assignment μ, by carrying out a parallel substitution of each of the variables x with the constant $\mu(x)$. Such a simplified formula is referred to as the restriction of ϕ with respect to μ, and is denoted as $\phi_{\downarrow\mu}$.

We write $\mu \models \phi$ to indicate that the variable assignment μ satisfies or models ϕ, i.e. $\phi_{\downarrow\mu} \equiv \top$; we write $\psi \models \phi$ to indicate that all models satisfying ψ also satisfy ϕ; finally, we write $\models \phi$ to indicate that any model satisfies ϕ, i.e. that ϕ is a validity.

2.2 Symbolic Transition Systems

A system S is represented as a *symbolic transition system* (STS). An STS is a tuple $\langle X, X_o, I, T\rangle$, where X is a finite non-empty set of state variables, $X_o \subseteq X$ is the set of observable state variables, I is a formula over X defining the initial states, T is a formula over X, X' – with X' representing the state variables after one transition – defining the transition relation. In the following we assume that an STS $S \doteq \langle X, X_o, I, T\rangle$ is given. We simply write $S \doteq \langle X, I, T\rangle$ whenever $X_o = \emptyset$.

A *state* s is an assignment to the state variables X. We denote with s' the corresponding assignment to X'. The set of all possible states (state space) of S may be either finite or infinite (if any $x \in X$ has an infinite domain). The *observable part* $obs(s)$ of a state s is the projection of s on the subset X_o of observable state variables. Thus, $obs(s) = Proj(s, X_o)$. A *trace* of S is an infinite sequence $\pi = s_0, s_1, s_2, \ldots$ of states such that s_0 satisfies I and, for each $k \in \mathbb{N}$, $\langle s_k, s_{k+1} \rangle$ satisfies T. We write $\Pi(S)$ to represent the set of traces of S. The observable part of π is $obs(\pi) = obs(s_0), obs(s_1), obs(s_2), \ldots$. Given a trace $\pi = s_0, s_1, s_2, \ldots$ and an integer $k \in \mathbb{N}$, we denote with π^k the finite prefix $s_0, \ldots, s_k$ of π containing the first $k+1$ state pairs. We denote with $\pi[k]$ the $k+1$-th state s_k. We say that s is *reachable* (in k steps) in S iff there exists a trace $\pi \in \Pi(S)$ such that $s = \pi[k]$ for some $k \geq 0$. We say that the transition relation is *total* iff for each state s there exists a successor state, that is, $\forall s \exists t$ such that $T(s,t)$ is valid. Without loss of generality, in the following we assume that the plant is total, and consider infinite traces only. Finally, in this dissertation we assume, without loss of generality, that transition systems are synchronous, that is all variables are updated simultaneously.

A *parameterized STS* is an STS where some of the state variables retain their initial value throughout each transition. A variable x is a *parameter* in S iff $T \models (x = x')$. Intuitively, parameters are used to represent a specific configuration of a system, e.g. whether an option is active or not, or a possible threshold to be set at installation. Let $A \subseteq X$ be the set of parameters of S. An assignment μ to the parameters A induces a corresponding transition system $S_{\downarrow\mu}$ defined as $\langle X \setminus A, X_o \setminus A, I_{\downarrow\mu}, T_{\downarrow\mu} \rangle$.

2.3 Temporal Logic

We now introduce two types of temporal logics, Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL). These logics will be used to describe properties of the temporal behavior of systems. In our case they will be used to express desired properties w.r.t. timed failure propagation and diagnosability, and furthermore to check whether the system behavior actually exhibits those properties. The main difference between LTL and MTL is in the model of time. LTL has the implicit notion of one time unit passing during every transition; every variable thus is modified synchronously according to an implicit global clock. MTL on the other hand allows a more fine-grained description of just how much time passes during one transition, from 0 for instantaneous transition to any value $\delta \in \mathbb{R}_{\geq 0}$.

We will use MTL to study timed failure propagation, and LTL to study diagnosability. The reason is mainly that for diagnosability we build upon existing work that uses the LTL view of time, while for TFPGs we developed a new framework and were able to choose the more expressive MTL. More detailed motivations will be given in the respective chapters.

2.3.1 Linear Temporal Logic

First the syntax and semantics of Linear Temporal Logic with Past operators [Laroussinie et al., 2002] will be described. Atomic propositions may be built over $X \cup X'$, and represent sets of system states, next states, and their combinations. For example, the atom $x' - x \leq \delta$ models that the increase in the value of x after a transition is bounded by δ.

Given a set of atomic propositions AP, including the symbols $\top$ (true) and $\bot$ (false), LTL formulae are defined as follows, with $p \in AP$:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \mathsf{U} \phi_2 \mid \phi_1 \mathsf{S} \phi_2 \mid \mathsf{X}\phi \mid \mathsf{Y}\phi \mid \mathsf{Z}\phi$$

We use the following common abbreviations as syntactic sugar, with n being a fixed integer:

- $\mathsf{F}\phi \doteq \top \mathsf{U} \phi$
- $\mathsf{G}\phi \doteq \neg\mathsf{F}\neg\phi$

- $\mathsf{O}\phi \doteq \top\mathsf{S}\phi$
- $\mathsf{H}\phi \doteq \neg\mathsf{O}\neg\phi$
- $\mathsf{X}^n\phi \doteq \mathsf{X}\mathsf{X}^{n-1}\phi$ with $\mathsf{X}^0\phi \doteq \phi$
- $\mathsf{Y}^n\phi \doteq \mathsf{Y}\mathsf{Y}^{n-1}\phi$ with $\mathsf{Y}^0\phi \doteq \phi$
- $\mathsf{F}^{\leq n}\phi \doteq \phi \vee \mathsf{X}\phi \vee \cdots \vee \mathsf{X}^n\phi$
- $\mathsf{G}^{\leq n}\phi \doteq \phi \wedge \mathsf{X}\phi \wedge \cdots \wedge \mathsf{X}^n\phi$
- $\mathsf{O}^{\leq n}\phi \doteq \phi \vee \mathsf{Y}\phi \vee \cdots \vee \mathsf{Y}^n\phi$
- $\mathsf{H}^{\leq n}\phi \doteq \phi \wedge \mathsf{Z}\phi \wedge \cdots \wedge \mathsf{Z}^n\phi$

We interpret LTL formulae over infinite traces $\pi = s_0, s_1, s_2, \ldots$. We write $\pi_{[i,i+1]}$ for the interpretation $s_i(X) \cup s_{i+1}(X')$, so that the current state variables X are interpreted in s_i and the next state variables X' in s_{i+1}. The semantics of LTL is as follows:

- $\forall i \cdot \pi[i] \models \top$
- $\forall i \cdot \pi[i] \not\models \bot$
- $\pi, i \models p$, where p is an atomic proposition, iff $\pi_{[i,i+1]} \models p$.
- $\pi, i \models \phi \vee \psi$ iff $\pi, i \models \phi$ and $\pi, i \models \psi$, and similarly for conjunction and negation.
- $\pi, i \models \psi\mathsf{U}\phi$ iff there exists $j \geq i$ such that $\pi, j \models \phi$ and, for all $i \leq h < j$, $\pi, h \models \psi$.
- $\pi, i \models \psi\mathsf{S}\phi$ iff there exists $0 \leq j \leq i$ such that $\pi, j \models \phi$ and, for all $j < h \leq i$, $\pi, h \models \psi$.
- $\pi, i \models \mathsf{X}\phi$ iff $\pi, i+1 \models \phi$.

- $\pi, i \models \mathsf{Y}\phi$ iff $i > 0$ and $\pi, i-1 \models \phi$.
- $\pi, i \models \mathsf{Z}\phi$ iff $i = 0$ or $\pi, i-1 \models \phi$.

An LTL formula ϕ holds on a trace π (written $\pi \models \phi$) iff ϕ is true in π at step 0 (written $\pi, 0 \models \phi$).

2.3.2 Metric Temporal Logic

Metric Temporal Logic (MTL) [Koymans, 1990, Alur and Henzinger, 1993, Ouaknine and Worrell, 2008] is an extension of classical LTL, where the temporal operators are augmented with timing constraints. Given a set of atomic propositions AP, including the symbols $\top$ (true) and $\bot$ (false), MTL formulae are defined as follows, with $p \in AP$:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \mathsf{U}^I \phi_2 \mid \phi_1 \mathsf{S}^I \phi_2 \mid \mathsf{X}\phi \mid \mathsf{Y}\phi \mid \mathsf{Z}\phi$$

The intervals I can be (partially) open or closed, $[a, b]$, (a, b), $(a, b]$, $[a, b)$, with $a, b \in \{\mathbb{R}_{\geq 0} \cup +\infty\}$ and $a \leq b$. I is omitted if $I = [0, +\infty)$, and the resulting simplified operators are called *unconstrained*. We use the same syntactic sugar as for LTL.

In the present dissertation we work with transition systems, thus we adopt the interpretation of MTL over timed state sequences and not the alternative dense-time interpretation (see e.g. [Ouaknine and Worrell, 2008] for a comparison of the two semantics). When using MTL we thus assume the presence of a variable $\tau \in X$ with $\Delta(\tau) = \mathbb{R}_{\geq 0}$, associating each state with a time stamp. We assume that time advances monotonically, i.e. given a trace π, $\tau(s_i) \leq \tau(s_{i+1})$ for any state s_i.

The semantics of MTL is defined as follows:

- $\forall i \cdot \pi[i] \models \top$
- $\forall i \cdot \pi[i] \not\models \bot$

- $\pi, i \models p$, where p is an atomic proposition, iff $\pi_{[i,i+1]} \models p$.
- $\pi, i \models \phi \vee \psi$ iff $\pi, i \models \phi$ and $\pi, i \models \psi$, and similarly for conjunction and negation.
- $\pi, i \models \psi \mathsf{U}^I \phi$ iff there exists $j \geq i$ such that $\tau_j - \tau_i \in I$, $\pi, j \models \phi$ and, for all $i \leq h < j$, $\pi, h \models \psi$.
- $\pi, i \models \psi \mathsf{S}^I \phi$ iff there exists $j \leq i$ such that $\tau_i - \tau_j \in I$, $\pi, j \models \phi$ and, for all $j < h \leq i$, $\pi, h \models \psi$.
- $\pi, i \models \mathsf{X}\phi$ iff $\pi, i+1 \models \phi$.
- $\pi, i \models \mathsf{Y}\phi$ iff $i > 0$ and $\pi, i-1 \models \phi$.
- $\pi, i \models \mathsf{Z}\phi$ iff $i = 0$ or $\pi, i-1 \models \phi$.

As with LTL, an MTL formula ϕ holds on a trace π (written $\pi \models \phi$) iff ϕ is true in π at step 0.

An MTL (and LTL) formula of the form $\mathsf{G}\phi$ where ϕ does not contain any temporal operators is called an invariant property. ϕ is interpreted on every single state of a trace, and the property is falsified if any such state does not satisfy ϕ.

2.4 Symbolic Model-Checking

For both LTL and MTL, the problem of deciding whether a formula ϕ holds on all traces of a given system S, written $S \models \phi$ (ϕ holds in S), is called *model-checking* [Clarke et al., 1999]. Specifically, we use *symbolic* model-checking, which uses a logical formalism to represent the characteristic functions of sets of states and transitions [McMillan, 1993]. Many efficient algorithms and implementations of symbolic model-checkers exist, e.g. [Biere et al., 1999, 2002, Eén and Sorensson, 2003, McMillan, 2003, Bradley, 2011, Claessen and Sörensson, 2012, Cimatti and Griggio, 2012, Cimatti et al., 2014].

The implementations of the algorithms presented in this dissertations rely on the model-checker nuXmv [Cavada et al., 2014], which in turn uses Satisfiability Modulo Theories (SMT) [Barrett et al., 2009] to reason over (infinite-state) transition systems. SMT is an extension of standard Boolean satisfiability, where formulae are expressed in a combination of first-order theories. For instance, linear arithmetic over the rationals allows expressions such as $(b \wedge 3.4 * t > r)$, where b is a Boolean variable and t and r are rationals. The SMT satisfiability checking problem then consists in finding an assignment to the variables that make the formula true. As opposed to the SAT problem, in SMT the set of models satisfying a formula may be infinite.

Finally we note the recent development of specialized algorithms for parametric model-checking, which can be used to compute all parameter configurations that satisfy a certain property [Cimatti et al., 2008, André et al., 2009, Cimatti et al., 2013a, Bittner et al., 2014a, Bozzano et al., 2015d]. Formally, the parameter synthesis problem takes as input a transition system $S := \langle X, X_o, I, T \rangle$, a set of parameters $A \subseteq X$, and a formula ϕ in temporal logic, and returns the set $\{\mu \in 2^A \mid S_{\downarrow\mu} \models \phi\}$. We will use standard model-checking for verification problems and parametric model-checking for synthesis problems.

Chapter 3

Timed Failure Propagation Graphs

The first step in developing an FDIR architecture is to identify the faults and their effects on the system, i.e. to identify what exactly the system has to be protected against. Only then one has a basis for deciding what kind of monitors and recovery procedures are necessary. To this aim, the standard failure analyses are Fault Tree Analysis (FTA) [Vesely et al., 1981] and Failure Modes and Effects Analysis (FMEA) [McDermott et al., 1996]. These techniques however don't have a comprehensive support for timing of failure propagations and are specialized to specific discrete analyses that make it difficult to obtain a global integrated picture of the overall failure behavior of a system. This in turn makes it difficult to develop a coherent set of detailed FDIR requirements and to check whether a given FDIR architecture is able to handle all possible faults and their propagation effects.

To address these issues in current practice, *Timed Failure Propagation Graphs* (TFPGs) were recently investigated as an alternative failure analysis framework in the FAME R&D project of the European Space Agency [European Space Agency, 2011, FAME, 2016, Bittner et al., 2014b][1]. TFPGs are labeled directed graphs that represent the propagation of failures in a system, including information on timing delays and mode con-

[1]The initial work on the contributions presented in this chapter was performed within the FAME project.

straints of individual propagation links. They have been studied and used in practice since the early 1990s [Misra et al., 1992, Abdelwahed et al., 2009], primarily as a way to deploy diagnosis systems. They can be seen as an abstract representation of a corresponding dynamic system of greater complexity, describing the occurrence of failures, their direct effects at a local level, and the corresponding consequences over time on other parts of the system. TFPGs are a very rich formalism: they allow to model Boolean combinations of basic faults, intermediate events, and transitions across them, possibly dependent on the operational modes of the system, and to express constraints over the delays between individual events. They convey thus qualitative and quantitative information on the temporal ordering of failures and their propagation effects, integrating in a single artefact several features that are specific to either FMEA or FTA, enhanced with timing information.

TFPG validation with respect to system models TFPGs are a promising technology that can improve the way FDIR is designed. Still, several key issues need to be dealt with to allow a broader adoption. First we consider the situation where a TFPG is built manually by an engineer, similar to the manual process of performing FMEA or FTA. Just as it is difficult to get fault trees and failure mode and effect tables right, it is difficult to build TFPGs, possibly even more so as we combine several of their features and furthermore add timing information. There is therefore a clear need for a comprehensive framework that validates TFPGs w.r.t. a more detailed model representing the system's dynamic behavior. Specifically, we need techniques to make sure that no important failure behavior that is possible in the system is overlooked (i.e. not modeled) in the TFPG. Likewise, we want to make sure that the TFPG contains as few spurious behaviors as possible, even though it might be impossible to exclude all such behaviors, due to the approximative nature of the TFPGs. Note that we are interested in dynamic system models as opposed to only structural ones, in order to capture emergent faulty behavior caused by the dynamic interaction between different parts of the system and their evolution over time.

To this aim we first define desirable formal characteristics of TFPGs as abstractions of a given transition system: *completeness* guarantees that all failure propagations possible in the system are captured by the TFPG; *edge tightness* guarantees that the time and mode constraints of propagations are as accurate as possible. We furthermore develop corresponding verification problems and map them to the framework of temporal logic model-checking. If the properties are violated, diagnostic information for debugging is provided.

TFPG synthesis from system models Second, we need an efficient way to directly and automatically derive TFPGs from corresponding system models. Developing TFPGs by hand is laborious and error-prone. In some cases it might still be the preferred way to create them, for instance when it is possible to leverage the results of already performed safety analyses. However, when such previous results are not available and the engineer that needs to perform the analysis does not have a sufficiently deep understanding of the system's behavior under faults or possibly not enough time to execute the analysis by hand, then a better way to obtain them could be to generate them automatically from a corresponding system model. Obviously the main burden then lies on the system modeler, but it is arguably easier to create a model that specifies the behavior of individual parts of the system, how they interact, and how they can fail locally, than it is to directly model the failure behavior at subsystem and system level that *emerges* from these local behaviors and interactions. Such emergent behavior is not always obvious when looking at individual parts of the system under analysis, and this problem increases with increased system complexity.

Based on the formal framework developed for validating given TFPGs against system models, we therefore propose a set of algorithms to *automatically* derive TFPGs from the corresponding system models in a way that guarantees by construction a number of formal properties. What an engineer needs to provide as an input are the system model, the set of failure modes, and the set of discrepancies or feared events and monitors that should be included in the end result.

Contributions and chapter structure The chapter is structured as follows. After providing some further background on TFPGs and a running example in Section 3.1, three main sets of contributions are presented.

1. First, we provide a formal framework for treating TFPGs as abstractions of a corresponding transition system. For this a trace-based semantics for TFPGs is introduced in Section 3.2, allowing us to compare behaviors compatible with the TFPG constraints to behaviors possible in the system. Based on this semantics, in Section 3.3 a way to map system traces to TFPG traces is introduced and two important properties of TFPGs as abstractions of system behavior are defined. This abstraction framework is the basis for the next two contributions.

2. Second, we show in Section 3.4 how verification of these TFPG constraints on system traces can be reduced to a model-checking problem, thus making it possible to use off-the-shelf (and hence state-of-the-art) verification technology. With this we address the validation problem mentioned above.

3. Third, we describe in Section 3.5 a way to automatically derive a TFPG from a transition system. The algorithm is structured in

three parts: generation of an initial verbose graph topology (Section 3.5.1), simplification of graph structure for improved readability (Section 3.5.2), and tightening of edge parameters for obtaining accurate propagation constraints (Section 3.5.3). With this we address the synthesis problem mentioned above.

The chapter also describes our implementation of the developed algorithms in Section 3.6 and provides a comprehensive experimental evaluation in Section 3.7. Related work is discussed in Section 3.8, and Section 3.9 summarizes the chapter and gives an outlook on future work on the topic of TPFGs.

The core results of the chapter were published in [Bittner et al., 2016c] (trace-based semantics, behavioral validation, and edge tightening) and in [Bittner et al., 2016b] (graph synthesis and simplification). The integration of the implementation inside the xSAP toolkit has been described in [Bittner et al., 2016a], and the integration in the FAME development process in [Bittner et al., 2014b].

3.1 Background

In this section we introduce the classical definition of TFPGs as well as a running example that will be used for the rest of the chapter. TFPGs – first described in [Misra et al., 1992, Misra, 1994] – are directed graph models where nodes represent *failure modes* (root events of failure propagations) and *discrepancies* (deviations from nominal behavior caused by failure modes). Edges model the temporal dependency between the nodes. They are labeled with propagation *delay bounds*, and *system modes* indicating the system configurations in which the propagation is possible. TFPGs are formally defined as follows.

Definition 1 (TFPG)**.** *A* TFPG *is a structure* $G = \langle F, D, E, M, ET, EM, DC \rangle$, *where:*

- F *is a non-empty set of failure modes;*
- D *is a non-empty set of discrepancies;*
- $E \subset V \times V$ *is a non-empty set of edges connecting the set of nodes* $V = F \cup D$*;*
- M *is a non-empty set of system modes (we assume that at each time instant the system is in precisely one mode);*
- $ET : E \to I$ *is a map that associates every edge in* E *with a time interval* $[t_{min}, t_{max}] \in I$ *indicating the minimum and maximum propagation time on the edge, with* $I \in \mathbb{R}_{\geq 0} \times (\mathbb{R}_{\geq 0} \cup \{+\infty\})$ *and* $t_{min} \leq t_{max}$*;*
- $EM : E \to 2^M$ *is a map that associates to every edge in* E *a set of modes in* M *(we assume that* $EM(e) \neq \emptyset$ *for every edge* $e \in E$*);*
- $DC : D \to \{\texttt{AND}, \texttt{OR}\}$ *is a map defining the discrepancy type;*

Failure modes never have incoming edges. All discrepancies must have at least one incoming edge and be reachable from a failure mode node. Circular paths are possible – except self-loops or zero-delay loops that would allow discrepancies to activate independently from failure mode nodes. We use `OR`*(G) and* `AND`*(G) to indicate the set of OR nodes and AND nodes of a TFPG G, respectively,* $D(G)$ *to indicate all discrepancies, and* $F(G)$ *to indicate all failure modes.*

The running example *ForgeRobot* describes a robot working in a hypothetical industrial forge. The robot is either in standby in a safe area, or performs work in a critical area that has high heat levels. It moves around using its locomotion facilities. To prevent overheating in the critical area,

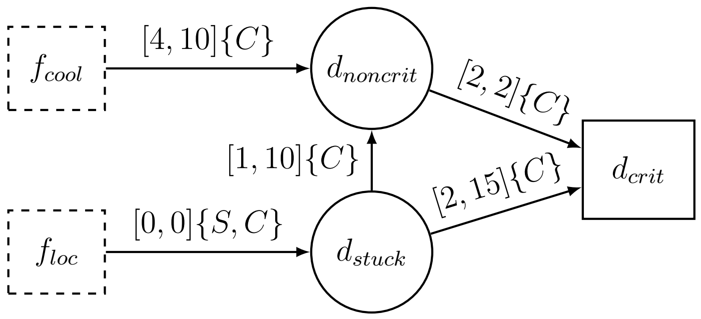


Figure 3.1: TFPG for the ForgeRobot example. Dotted boxes are failure mode nodes, solid boxes `AND` nodes, and circles `OR` nodes.

a cooling system is used. The TFPG in Figure 3.1 shows possible failures of the robot and their effects over time. Two modes are used to differentiate the operational context: S for safe area, and C for critical area. The locomotion drive of the robot can fail (f_{loc}), causing the robot to be stuck (d_{stuck}). The cooling system can fail (f_{cool}), decreasing the performance of heat protection. f_{cool} and d_{stuck} can both independently cause a non-critical overheating of the robot ($d_{noncrit}$) in mode C. In case both happen, they cause a critical overheating (d_{crit}). The fact that failure mode labels start with f and discrepancy labels with d is just a convention used for this particular example. The time ranges on the propagation edges represent the different propagation speeds influenced by variable amount of workload and of heat in the critical area.

According to the semantics of TFPGs [Abdelwahed et al., 2009], a TFPG node is activated when a failure propagation has reached it. An edge $e = (v, d)$ is active iff the source node v is active and $m \in EM(e)$, where m is the current system mode. A failure propagates through $e = (v, d)$ only if e is active throughout the propagation, that is, up to the time d activates. For an `OR` node d and an edge $e = (v, d)$, once e becomes active at time t, the propagation will activate d at time t', where $tmin(e) \leq t' - t \leq tmax(e)$. Activation of an `AND` node d will occur at time t' if every edge $e = (v, d)$ has been activated at some time t, with $tmin(e) \leq t' - t$; for at least one such edge e we must also have $t' - t \leq tmax(e)$, i.e. the upper bound can be exceeded for all but one edge. If an edge is deactivated any time during the propagation, due to mode switching, the propagation stops. Links are assumed memory-less, thus failure propagations are independent of any (incomplete) previous propagation.

A maximum propagation time of $t_{max} = +\infty$ indicates that the propagation across the respective edge can be delayed indefinitely, i.e. it might never occur at all. This is a useful over-approximation when the real t_{max} value is not available; it is also necessary when the propagation depends on some unconstrained input or other dynamics not captured by the TFPG. An alternative interpretation is described in [Bozzano et al., 2015b], prescribing that the propagation will eventually occur, but without guarantees on when exactly. It could be an interesting aspect for future work to support also this semantics, possibly as an alternative as the two interpretations are conflicting.

3.2 Trace-Based Semantics

To enable the mapping of system traces to TFPG traces, we define TFPGs as transition systems, whose paths describe failure propagations as timed sequences of failure mode and discrepancy occurrences and mode switches. We first define a TFPG transition system over TFPG nodes, modes, and time delays. Then, given a TFPG G defined over the same nodes and modes, we enforce the propagation constraints represented by the edges and the discrepancy classes. The idea is not to model the propagations per se, but rather to observe the occurrence of discrepancy events and mode switches and to check whether these activations (or absence thereof) are in violation of the TFPG constraints.

With this modeling approach it is not always possible to determine along which exact path a propagation occurred, e.g. in the running example it might not be clear along which path $d_{noncrit}$ was activated, if the preconditions of all incoming edges are satisfied. However, it is always possible to check whether a trace satisfies the propagation constraints. This framework enables the comparison of system behaviors and TFPG behaviors and the abstraction of system behaviors in terms of TFPG behaviors.

Definition 2 (TFPG Transition System)**.** *Given are a set of failure mode variables F, a set of discrepancy variables D, and a set of system modes M. A* TFPG Transition System *is a tuple $S_{tfpg} = \langle X, I, T \rangle$ such that:*

- *$X = F \cup D \cup M \cup \tau$, with $\Delta(x) = \{\top, \bot\}$ for $x \in F \cup D \cup M$ and $\Delta(\tau) = \mathbb{R}_{\geq 0}$;*
- *$I(X) = \phi_{modes}(M) \wedge \tau = 0$;*
- *$T(X, X') = \phi_{modes}(M') \wedge \bigwedge_{x \in \{F \cup D\}} (x \rightarrow x') \wedge (\tau \leq \tau') \wedge ((\bigvee_{x \in \{F \cup D \cup M\}} (x \neq x')) \rightarrow (\tau = \tau'))$*

where $\phi_{modes}(M) \equiv \bigwedge_{m \in M} (m \leftrightarrow \bigwedge_{n \in \{M \setminus m\}} \neg n)$. A trace π of a TFPG transition system is called a TFPG trace*. We write $TS(G)$ to indicate the TFPG transition system derived from the nodes $F \cup D$ and modes M of the TFPG G.*

For $x \in \{F \cup D\}$, $x = \top$ indicates that the node x is active in the current state, whereas for $x \in M$ it means that the system is currently in mode x. Formula $\phi_{modes}(M)$ states that the system is in precisely one mode at any time. The transition relation enforces that TFPG nodes stay active once activated, and that time advances monotonically. Note that it also enforces that time does not pass during discrete switches. We use this modeling assumption to pin-point the moment in time when any discrete change in TFPG state occurred, in order to unambiguously establish the delay between such events.

With respect to time we regard TFPG traces as timed state sequences and thus use the MTL model of time. As opposed to the LTL view which uses an atomic unit of time that elapses during each transition, this approach will allow not only a much more fine-grained modeling of time, but (and perhaps more importantly in practice) it will allow to compactly represent dynamics with few discrete changes over time but large amounts of time elapsing. This is crucial for performance reasons in industrial applications, and will be discussed on an example in Chapter 5.

We define now under which conditions a trace of a TFPG transition system S_{tfpg} satisfies the constraints of a given TFPG. We use the notation $\mu(e) = \bigvee_{m \in EM(e)} m$ to indicate the system modes that are supported by an edge $e \in E$.

Definition 3 (OR-node satisfaction)**.** *Given a TFPG G, we say that a trace π of $TS(G)$ satisfies the constraints of an* `OR` *node $d \in D$ of G iff for any state $\pi[k]$:*

A. $(\pi[k] \models d) \rightarrow \exists j \leq k \cdot ((\pi[j] \models d) \wedge \exists e = (v, d) \in E\ \exists i \leq j \cdot ((\tau_j - \tau_i \geq tmin(e)) \wedge \forall i \leq l \leq j \cdot (\pi[l] \models (v \wedge \mu(e)))))$.

B. $\forall e = (v, d) \in E \cdot (\neg \exists i \leq k \cdot ((\tau_k - \tau_i > tmax(e)) \wedge \forall i \leq j \leq k \cdot \pi[j] \models (v \wedge \mu(e) \wedge \neg d)))$.

Condition A of Definition 3 states that if d is active in $\pi[k]$ then it must have been activated at some previous point $\pi[j]$ after some edge leading to d was active for at least the respective t_{min}, starting from $\pi[i]$, up to $\pi[j]$ where d became active. Condition B instead states that for no edge e the propagation can be delayed for more than the respective t_{max}. When $tmax(e) = +\infty$ for some edge $e = (v, d)$, then the existentially quantified part of Condition B is always false and thus cannot falsify the condition as a whole, whose satisfaction thus effectively depends on edges whose upper bound is not $+\infty$.

Definition 4 (AND-node satisfaction)**.** *Given a TFPG G, we say that a trace π of $TS(G)$ satisfies the constraints of an* `AND` *node $d \in D$ of G iff for any state $\pi[k]$:*

A. $(\pi[k] \models d) \rightarrow \exists j \leq k \cdot ((\pi[j] \models d) \wedge \forall e = (v, d) \in E\ \exists i \leq j \cdot ((\tau_j - \tau_i \geq tmin(e)) \wedge \forall i \leq l \leq j \cdot (\pi[l] \models (v \wedge \mu(e)))))$.

B. $\exists e = (v, d) \in E \cdot (\neg \exists i \leq k \cdot ((\tau_k - \tau_i > tmax(e)) \wedge \forall i \leq j \leq k \cdot \pi[j] \models (v \wedge \mu(e) \wedge \neg d)))$.

Condition A of Definition 4 states that if d is active in $\pi[k]$ then it has been activated at some previous point $\pi[j]$ after all edges leading to d were active for at least the respective t_{min}, each starting from some individual $\pi[i]$, up to $\pi[j]$ where d became active. Condition B instead states that at least for one edge e the propagation must respect the respective t_{max} bound. The difference in Definitions 3 and 4 is in the quantifiers over the edges, corresponding to the semantics of the `OR` and `AND` nodes. When $tmax(e) = +\infty$ for some edge e, then the existentially quantified part of Condition B is always false too; however here it has the effect, due to the first existential quantifier, that the whole condition is always satisfied whenever any $e = (v, d)$ has $tmax(e) = +\infty$.

Definition 5 (TFPG satisfaction)**.** *Given a TFPG G, we say that a trace π of $TS(G)$* satisfies *G iff π satisfies, for all $d \in D$, the conditions of Definition 3 or Definition 4, depending on whether d is an* `OR` *node or* `AND` *node, respectively.*

In the following sections we use $\Pi^*(G)$ to indicate all possible traces of $TS(G)$, and we use $\Pi(G) \subseteq \Pi^*(G)$ to indicate all traces of $TS(G)$ that satisfy G as per Definition 5.

From the definitions above we can also abstract the timing part and derive purely Boolean constraints over the node activations: a trace $\pi \in$

$TS(G)$ satisfies the Boolean constraints as encoded by G iff $\forall k \in \mathbb{N} \cdot \pi[k] \models \phi_{bool}(G)$, where $\phi_{bool}(G) := \bigwedge_{d \in \texttt{OR(G)}}(d \rightarrow \bigvee_{(v,d) \in E} v) \wedge \bigwedge_{d \in \texttt{AND(G)}}(d \rightarrow \bigwedge_{(v,d) \in E} v)$.

TFPG satisfaction is based on *local* node activation constraints. According to Definition 5, a TFPG trace satisfies the TFPG constraints if all individual nodes are activated according to the respective local constraints, and if no node activation is delayed beyond the respective local upper bounds on propagation delay. Note that failure mode nodes $fm \in F$ need not be considered, since their activation is completely unconstrained w.r.t. the other TFPG elements. As a consequence of using local constraints, pin-pointing violations to TFPG satisfaction becomes straightforward and makes it easier to fix the graph, the edge constraints, or the system model.

This trace-based semantics for TFPGs closely follows the original semantics of TFPGs as described earlier. Its adequacy has been validated in an industrial case study in the FAME project [Bittner et al., 2014b], and in the case studies described in Chapter 5.

3.3 System Abstraction

The core idea of this chapter is to abstract systems using TFPGs, by associating system traces with TFPG traces. To this aim, we first define TFPG elements of interest (failure modes, discrepancies and modes) in terms of system properties, as follows.

Definition 6 (TFPG Association Map). *Given a set of failure mode variables F, a set of discrepancy variables D, a set of system modes M, a time-stamp variable τ, and a system model S_{sys}, a* TFPG Association Map *is w.l.o.g. an injective function Γ that associates every variable $x \in \{F \cup D \cup M\}$ with a Boolean predicate γ over the state variables X of*

S_{sys}, written $\gamma_x(X)$, or simply γ_x when the reference to X is clear from the context, and τ with a variable $x \in X$, representing the state time-stamps in the system, with $\Delta(x) = \mathbb{R}_{\geq 0}$. Given an edge $e \in E$, we use the short form $\gamma_{\mu(e)}$ for $\bigvee_{m \in EM(e)} \gamma_m$. Following the assumption of modes partitioning the state space, we assume $\gamma_{m1} \wedge \gamma_{m2} \equiv \bot$ for any $m1, m2 \in M \cdot m1 \neq m2$, and that for any state s of S_{sys} there exists $m \in M$ such that $s \models \gamma_m$.

These maps are specified by the user. They only associate TFPG variables with system properties and do not refer to the propagation constraints encoded by edges and discrepancy classes. For instance, in the running example the discrepancy $d_{noncrit}$ may be defined by the expression $robot.temperature \geq 10$.

We also allow the specification of *virtual discrepancies*, which are auxiliary TFPG nodes used to express the temporal relationships among the other user-defined nodes, but which don't have a directly corresponding property in the system. For instance, when the goal of TFPG modeling is to relate input faults of a component directly to output faults, but the Boolean relationship cannot be encoded by the discrepancy class of output discrepancies, then a virtual discrepancy might be used without the need to bind it to an internal state property of the component. These discrepancies will also be essential for the synthesis algorithm described in Section 3.5.

Definition 7 (Virtual Discrepancy)**.** *Given a TFPG G, a system model S, and an association map Γ that is total w.r.t. $F(G)$ but partial w.r.t. $D(G)$, a* virtual discrepancy *is a node $d \in D(G)$ that is not in the domain of Γ. Instead of being associated, w.r.t. S, to a Boolean expression γ_d, such discrepancies are associated to temporal expressions over the traces of S, following the structure of G. For `OR` nodes the associated expression is $\gamma_d := \bigvee_{(v,d) \in E} \mathsf{O}\gamma_v$, and for `AND` nodes it is $\gamma_d := \bigwedge_{(v,d) \in E} \mathsf{O}\gamma_v$. It is assumed that no $v \in D(G)$, where $(v,d) \in E$, is itself a virtual discrepancy.*

Intuitively, w.r.t. system traces, a virtual discrepancy activates at the same instant as one or all of its predecessors in the graph activate, depending on its node type.

When interpreting a system trace in terms of TFPG primitives we are interested in the points in the trace where failure modes occur, discrepancies become true or the system mode changes, and in the order and time delay between these events. Definition 8 defines a mapping from system traces to TFPG traces that guarantees that the order and timing of TFPG events is the same as in the system trace.

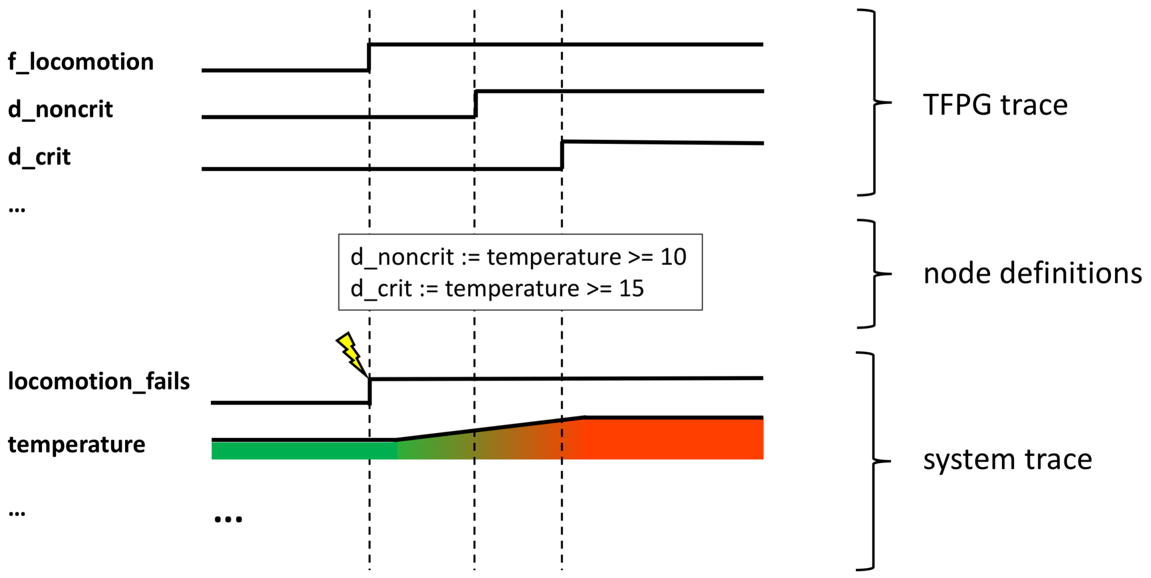


Figure 3.2: Example of trace abstraction for ForgeRobot. Square signals are used to model Boolean values over time.

Definition 8 (Trace Abstraction)**.** *Given a system model* $S_{sys} = \langle X, I, T \rangle$*, a TFPG transition system* $S_{tfpg} = \langle X_G, I_G, T_G \rangle$ *with* $X_G = \{F \cup D \cup M \cup \{\tau\}\}$*, and a TFPG association map* Γ *defining the symbols in* X_G *based on predicates interpreted over* X*, we define the* trace abstraction ζ_Γ *of a system trace* π *producing an abstract TFPG trace* π' *of* S_{tfpg}*, written* $\pi' = \zeta_\Gamma(\pi)$*, as follows:*

- $\forall x \in \{F \cup D\} \forall k \in \mathbb{N} \cdot ((\pi'[k] \models x) \leftrightarrow \exists i \leq k \cdot (\pi[i] \models \gamma_x))$
- $\forall x \in \{M \cup \tau\} \forall k \in \mathbb{N} \cdot (x(\pi'[k]) = \gamma_x(\pi[k]))$

The variables $x \in \{F \cup D\}$ effectively behave as history monitors for their associated expression, whereas the variables $x \in \{M \cup \{\tau\}\}$ are evaluated for every individual state.

Given a system trace π, we assume that for every $x \in \{F \cup D \cup M\}$ and every point k it holds that $\gamma_x(\pi[k]) \neq \gamma_x(\pi[k+1]) \rightarrow \gamma_\tau(\pi[k]) = \gamma_\tau(\pi[k+1])$, i.e. time does not pass when the truth value of the predicate defining x changes. This allows to unambiguously measure the delay between discrete TFPG-related events in the system trace, and is similar in spirit to discrete switches in timed or hybrid automata, during which time doesn't pass. We also assume that the system is at each time instant in precisely one mode. These assumptions guarantee that the abstract traces of Definition 8 satisfy the constraints of Definition 2. An example trace abstraction for the ForgeRobot model is shown in Figure 3.2.

Completeness and Tightness

The notion of completeness of a TFPG G reflects the fact that all possible failure propagations in S are also modeled by G, in other words, that the abstraction of every system trace satisfies the constraints of G. In a certain sense completeness asks whether the behaviors allowed by the TFPG are an over-approximation of the failure propagation behaviors that are possible in the system model. Conversely, a TFPG is not complete if some failure propagation pattern exists on some system trace that is not captured by it. In the running example such a trace would be, e.g., one where first f_{loc} happens and then $d_{noncrit}$, without any other node activating, which violates the TFPG constraints.

Definition 9 (Completeness)**.** *Given a system model S, a TFPG G, and a TFPG association map Γ connecting the two, we say that G is* complete *w.r.t. S* iff *for every trace π of S, its abstraction $\zeta_\Gamma(\pi)$ satisfies G, i.e. $\zeta_\Gamma(\pi) \in \Pi(G)$.*

Conversely to the notion of completeness, it is legitimate to ask whether each failure propagation modeled by a TFPG can actually take place in the corresponding system. This question is misleading, since a TFPG is naturally an over-approximation of an underlying system. System modes for instance are completely unconstrained in a TFPG, but in realistic cases this is not true in the system. Instead we propose to study the property of tightness, i.e. whether certain parameters of the TFPG can be reduced without breaking its completeness. Specifically, we are interested in tightening the propagation intervals and the modes.

Definition 10 (Edge Tightness)**.** *Given are a system model S, a TFPG G, an association map Γ, and an edge $e \in E$ of G. We say that $tmin(e)$ is tight* iff *there is no $r > tmin(e)$ such that G is complete w.r.t. S with $tmin(e) := r$ and all other parameters of G remaining the same. We say that $tmax(e)$ is tight* iff *there is no $r < tmax(e)$ such that G is complete w.r.t. S with $tmax(e) := r$ and all other parameters of G remaining the same. We say that $EM(e)$ is tight* iff *there exists no $m \in EM(e)$ such that G is complete w.r.t. S with $EM(e) := EM(e) \setminus m$ and all other parameters of G remaining the same. Finally, we say that the edge e is tight* iff *$tmin(e)$, $tmax(e)$, and $EM(e)$ are tight.*

We remark that this definition checks for the effect of single parameter changes. In the running example we might for instance check whether the propagation from f_{loc} to d_{stuck} really can occur in mode C, or whether some system behavior exists where the delay between f_{cool} and $d_{noncrit}$ is indeed 10 time units.

Completeness might be preserved when changing multiple parameters simultaneously. For instance, if a mode m on an edge e is dropped in which the propagation cannot occur at all and $tmax(e)$ is set to a finite value instead of $+\infty$ that is a correct bound for the propagation in the remaining modes, completeness is preserved, while just reducing $tmax(e)$ would break it.

3.4 Behavioral Validation

In this section we describe a method to check whether a TFPG is a complete and tight abstraction of a corresponding system. First we show how completeness can be checked by reduction to an MTL model-checking problem; then we show how tightness can be checked using a number of completeness checks.

3.4.1 Completeness

We show now how TFPG completeness can be reduced to a model-checking problem of an MTL formula over the original system. A composition of the system model and the TFPG transition system will not be necessary, and we only need to work with the system traces.

Theorem 1 and Theorem 2 provide partial proof obligations (for `OR` and `AND` nodes, respectively) to check whether the constraints of individual nodes are satisfied on a system trace. Theorem 3 then combines them into a check for overall completeness. The intuition behind the theorems is that the proof obligations can be derived from the definitions of OR-node satisfaction and AND-node satisfaction via the semantics of temporal operators and the mappings of Definition 8.

Theorem 1. *Given a system model S, an association map Γ relating S to* *a given TFPG G, and an* **OR** *node d of G, we define the following proof obligations:*

1. $\psi_{\texttt{OR}\cdot A}(d,\Gamma) := \mathsf{G}((\mathsf{O}\gamma_d) \rightarrow \mathsf{O}((\mathsf{O}\gamma_d) \wedge \bigvee_{e=(v,d)\in E}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \mathsf{S}^{\geq tmin(e)} (\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)})))$
2. $\psi_{\texttt{OR}\cdot B}(d,\Gamma) := \mathsf{G}\neg(\bigvee_{e=(v,d)\in E}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \wedge \neg(\mathsf{O}\gamma_d) \mathsf{S}^{>tmax(e)} ((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \wedge \neg(\mathsf{O}\gamma_d)))$

For a trace π of S, $\zeta_\Gamma(\pi)$ satisfies the constraints of d, as per Definition 3, iff *$\pi \models \psi_{\texttt{OR}\cdot A}(d,\Gamma)$ and $\pi \models \psi_{\texttt{OR}\cdot B}(d,\Gamma)$.*

Proof. From Definition 8 we know that: for any node $v \in \{F \cup D\}$ we have $\forall k \in \mathbb{N}\cdot\ \zeta_\Gamma(\pi)[k] \models v$ *iff* $\exists i \leq k \cdot \pi[i] \models \gamma_v$; for all edges $e \in E$ we have $\forall k \in \mathbb{N}\cdot\ \zeta_\Gamma(\pi)[k] \models \mu(e)$ *iff* $\pi[k] \models \gamma_{\mu(e)}$. Based on this, the FOL semantics of TFPG trace validity w.r.t. an OR node $d \in D$ of G can be derived from the MTL formulae as follows, showing that π satisfies $\psi_{\texttt{OR}\cdot A}(d,\Gamma)$ and $\psi_{\texttt{OR}\cdot B}(d,\Gamma)$ *iff* $\zeta_\Gamma(\pi)$ satisfies the constraints of d. For $\psi_{\texttt{OR}\cdot A}(d,\Gamma)$ this is shown as follows:

- $\pi \models \mathsf{G}((\mathsf{O}\gamma_d) \rightarrow \mathsf{O}((\mathsf{O}\gamma_d) \wedge \bigvee_{e=(v,d)\in E}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \mathsf{S}^{\geq tmin(e)} (\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)})))$

  iff, by semantics of temporal operators,

- $\forall k \in \mathbb{N} \cdot ((\exists i \leq k \cdot \pi[i] \models \gamma_d) \rightarrow \exists i \leq k \cdot ((\exists j \leq i \cdot \pi[j] \models \gamma_d) \wedge \exists e = (v,d) \in E \exists j \leq i \cdot (\tau_i - \tau_j \geq tmin(e) \wedge \forall j \leq l \leq i \cdot ((\exists n \leq l \cdot \pi[n] \models \gamma_v) \wedge (\pi[l] \models \gamma_{\mu(e)})))))$

  iff, by the above-mentioned notions following Definition 8,

- $\forall k \in \mathbb{N} \cdot ((\zeta_\Gamma(\pi)[k] \models d) \rightarrow \exists i \leq k \cdot ((\zeta_\Gamma(\pi)[i] \models d) \wedge \exists e = (v,d) \in E \exists j \leq i \cdot (\tau_i - \tau_j \geq tmin(e) \wedge \forall j \leq l \leq i \cdot ((\zeta_\Gamma(\pi)[l] \models v) \wedge (\zeta_\Gamma(\pi)[l] \models \mu(e))))))$

which is equivalent to formula A of Definition 3. For $\psi_{\texttt{OR}\cdot B}(d,\Gamma)$ the derivation is done similarly as follows:

- $\pi \models \mathsf{G}\neg(\bigvee_{e=(v,d)\in E}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \wedge \neg(\mathsf{O}\gamma_d)\mathsf{S}^{>tmax(e)}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \wedge \neg(\mathsf{O}\gamma_d)))$

iff, by semantics of temporal operators,

- $\forall k \in \mathbb{N} \cdot \neg(\exists e = (v,d) \in E \exists i \leq k \cdot (\tau_k - \tau_i > tmax(e) \wedge \forall i \leq j \leq k \cdot ((\exists l \leq j \cdot \pi[l] \models \gamma_v) \wedge (\pi[j] \models \gamma_{\mu(e)}) \wedge \neg(\exists l \leq j \cdot \pi[l] \models \gamma_d))))$

iff, by the above-mentioned notions following Definition 8,

- $\forall k \in \mathbb{N} \cdot \neg(\exists e = (v,d) \in E \exists i \leq k \cdot (\tau_k - \tau_i > tmax(e) \wedge \forall i \leq j \leq k \cdot ((\zeta_\Gamma(\pi)[j] \models v) \wedge (\zeta_\Gamma(\pi)[j] \models \mu(e)) \wedge \neg(\zeta_\Gamma(\pi)[j] \models d))))$

which is equivalent to formula B of Definition 3. □

We now define the corresponding proof obligations for `AND` nodes.

Theorem 2. *Given a system model S, an association map Γ relating S to a given TFPG G, and an* `AND` *node d of G, we define the following proof obligations:*

1. $\psi_{\texttt{AND}\cdot A}(d,\Gamma) := \mathsf{G}((\mathsf{O}\gamma_d) \rightarrow \mathsf{O}((\mathsf{O}\gamma_d) \wedge \bigwedge_{e=(v,d)\in E}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)}\mathsf{S}^{\geq tmin(e)}(\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)})))$
2. $\psi_{\texttt{AND}\cdot B}(d,\Gamma) := \mathsf{G}\neg(\bigwedge_{e=(v,d)\in E}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \wedge \neg(\mathsf{O}\gamma_d)\mathsf{S}^{>tmax(e)}((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)} \wedge \neg(\mathsf{O}\gamma_d)))$

For a trace π of S, $\zeta_\Gamma(\pi)$ satisfies the constraints of d, as per Definition 4, iff $\pi \models \psi_{\texttt{AND}\cdot A}(d,\Gamma)$ *and* $\pi \models \psi_{\texttt{AND}\cdot B}(d,\Gamma)$.

Proof. The proof is symmetrical to the proof of Theorem 1, except that we use universal quantification to represent the semantics of $\bigwedge_{e=(v,d)\in E}$. □

Intuitively, $\psi_{\texttt{OR}\cdot A}(d, \Gamma)$ requires at least one edge $e = (v, d)$ to be active for at least $tmin(e)$ time units at the point where d activates, and $\psi_{\texttt{OR}\cdot B}(d, \Gamma)$ requires that a propagation along some edge $e = (v, d)$ cannot be delayed for more than $tmax(e)$ time units.

Based on Theorems 1 and 2, Theorem 3 formulates the proof obligation that a system trace must satisfy in order for the corresponding TFPG trace to satisfy a given TFPG.

Theorem 3. *Given a system model S, a TFPG G, and an association map Γ relating S to G, let $\Psi(G, \Gamma) := \bigwedge_{d \in \texttt{OR(G)}}(\psi_{\texttt{OR}\cdot A}(d, \Gamma) \wedge \psi_{\texttt{OR}\cdot B}(d, \Gamma)) \wedge \bigwedge_{d \in \texttt{AND(G)}}(\psi_{\texttt{AND}\cdot A}(d, \Gamma) \wedge \psi_{\texttt{AND}\cdot B}(d, \Gamma))$. Then, G is complete w.r.t. S* iff *$S \models \Psi(G, \Gamma)$.*

Proof. Definition 9 states that G is complete w.r.t. S *iff* for every trace π of S it holds that its abstraction $\zeta_\Gamma(\pi)$ satisfies G, i.e. $\zeta_\Gamma(\pi) \in \Pi(G)$. This on the other hand is true *iff*, following Definition 5, for every trace π of S it holds that $\zeta_\Gamma(\pi)$ satisfies, for every $d \in D$, the conditions of Definition 3 or Definition 4, depending on whether d is an `OR` node or `AND` node. From Theorem 1 (resp. Theorem 2) we know that, for a system trace π and an `OR` node (resp. `AND` node) $d \in D$, $\zeta_\Gamma(\pi)$ satisfies the conditions of Definition 3 (resp. Definition 4) *iff* $\pi \models \psi_{\texttt{OR}\cdot A}(d, \Gamma) \wedge \psi_{\texttt{OR}\cdot B}(d, \Gamma)$ (resp. $\pi \models \psi_{\texttt{AND}\cdot A}(d, \Gamma) \wedge \psi_{\texttt{AND}\cdot B}(d, \Gamma)$). □

Note that for a virtual discrepancy d, the corresponding γ_d is a temporal expression and not a purely Boolean one (see Definition 7), but this has no effect on the theorems and their proofs. Furhtermore, given an edge $e = (v, d)$, the subclauses relative to e in $\psi_{\texttt{OR}\cdot B}(d, \Gamma)$ and $\psi_{\texttt{AND}\cdot B}(d, \Gamma)$ are trivially false when $tmax(e) = +\infty$ and can be simplified accordingly.

3.4.2 Edge Tightness

Edge tightness can be reduced to a number of completeness checks. As per Definition 10, checking the tightness of an edge $e \in E$ amounts to verifying, individually for each parameter $tmin(e)$, $tmax(e)$, and $EM(e)$, if there exists a tighter assignment such that the accordingly modified TFPG G' is still complete w.r.t. S: $S \models \Psi(G', \Gamma)$. This can be done by searching over the range of possible tighter parameter assignments.

Note that tightness checks for some edge $e = (v, d)$ only require to evaluate the proof obligations affected by the parameter change. For instance, if d is an `OR` node and we check the tightness of $tmin(e)$, then only $\psi_{\texttt{OR}\cdot A}(d, \Gamma)$ needs to be evaluated. This is possible as by assumption of completeness of the original TFPG, all proof obligations related to individual discrepancies hold for the original TFPG, and as they do not refer to the constant $tmin(e)$ they will still hold.

3.5 Synthesis

In this section we describe a method to synthesize a TFPG in a fully automated way, starting from the following inputs:

- a system model;
- a set of failure mode and discrepancy nodes;
- an association map defining them.

Whereas before we assumed the TFPG to be given, possibly modeled by hand by an engineer, we would like to have now a procedure that takes these inputs, first computes the underlying graph of the TFPG, by analyzing the system traces, and subsequently tightens its edges, resulting in a complete and tight TFPG. For the running example this means that from just the nodes f_{cool}, f_{loc}, d_{stuck}, $d_{noncrit}$, and d_{crit} we want to automatically compute the TFPG shown in Figure 3.1.

Before defining more formally the goal of the synthesis procedure, note that what the edges of a TFPG represent, from a qualitative point of view, is the temporal ordering of the events they connect. The edge $(f_{cool}, d_{noncrit})$ means that first f_{cool} will happen, and then $d_{noncrit}$. In the system model we will thus have traces where first the event associated to f_{cool} happens, and after that the one associated to $d_{noncrit}$. Furthermore, according to the TFPG (and the system model) $d_{noncrit}$ might also follow after d_{stuck}, which itself must be preceeded by f_{loc}. Time bounds and mode labels then provide additional information about the basic qualitative relationships encoded by the plain graph.

The qualitative topology of a TFPG thus encodes a number of *precedence constraints* among its nodes, prescribing what other nodes a failure has to propagate through before reaching any particular discrepancy. In other words, they prescribe the sets of nodes that have to be activated before the given target discrepancy can be activated.

The goal of the synthesis procedure then should be to identify all precedence constraints from the system traces and encode them in the graph of the TFPG. A complete and tight TFPG could be trivially obtained also by instantiating every discrepancy as an `OR` node and connecting every failure mode node to every discrepancy, with tightened edges. This way however the information on the temporal ordering among events is lost, and the resulting TFPG would provide very limited insight (if any at all) into the system dynamics following a failure.

The core insight on which the proposed synthesis algorithm is based on is that this notion of precedence constraints matches another well-known notion in the field of failure analyses, that is the one of minimal cut-sets from fault tree analysis. In FTA the goal is to identify the sets of basic events (cut-sets) which can lead to a specific effect called *top-level event* (TLE). Of particular interest are *minimal cut-sets*, i.e. cut-sets whose basic events must all occur before the top-level event will occur; conversely, the TLE will not occur before all elements of a minimal cut-set have occurred. We report here the definition from [Bozzano et al., 2015d], adapted to our setting.

Definition 11 (Cut-Set). *Given are a transition system S and a set of events $EV = \langle e_1 \dots e_n \rangle$, where each event $e_i \in EV$ is defined by a Boolean formula interpreted over the state vector of S. Given an event $e \in EV$, a set $cs \subseteq EV \setminus e$ is a* cut-set *of e iff there exists a trace π of S for which $\exists k \in \mathbb{N}$ such that $\pi[k] \models e$ and $\forall e' \in EV \setminus e \cdot e' \in cs \Leftrightarrow \exists i \leq k \cdot \pi[i] \models e'$. A cut-set cs of e is* minimal *iff no proper subset of cs is a cut-set of e. We use $MCS(e, EV, S)$ as a short form to indicate the set of all minimal cut-sets of event e in S w.r.t. EV.*

In a TFPG transition system the events are the nodes being activated. In a system model they are the Boolean properties associated to the nodes by the association map becoming true. In the TFPG G of the running example, $\{f_{loc}, d_{stuck}\}$ is a minimal cut-set of $d_{noncrit}$, considering the traces in $\Pi(G)$. This means that, after the activations of f_{loc} and d_{stuck}, the node $d_{noncrit}$ can become active before any other node does, e.g. f_{cool} doesn't have to occur before $d_{noncrit}$. Figure 3.3 shows all minimal cut-sets of each discrepancy in the running example.

d_{stuck}	(f_{loc})
$d_{noncrit}$	(f_{cool}), (f_{loc}, d_{stuck})
d_{crit}	$(f_{loc}, d_{stuck}, d_{noncrit})$

Figure 3.3: Minimal cut-sets of all discrepancies in the running example.

Based on the notion of minimal cut-sets, we then define precedence constraints as follows.

Definition 12 (Precedence Constraints)**.** *Given are a transition system S, a set of events EV, and some $e \in EV$. The* precedence constraints *of e are the set of all of its minimal cut-sets w.r.t. EV and S. The precedence constraints of e are satisfied on some trace and state $\pi[k]$ iff all events of one of e's minimal cut-sets have occurred at some state j, with $0 \leq j \leq k$.*

The precedence constraints of e prescribe which events of EV must happen on the traces of S before e itself will occur. Note that satisfaction of these conditions does not imply that the effect inexorably follows. On the level of the TFPG this means that a propagation might not be completed due to dynamics within the system model that are invisible from the point of view of the TFPG. Such cases are modeled in the TFPG by reducing the mode labels of edges, if the propagation never occurs in one of the possible modes, or setting specific t_{max} parameters to $+\infty$ (see Section 3.1).

Based on the notion of precedence constraints, we now define the property of *graph correctness* which we want the synthesized TFPG to have. We rely on Γ to map sets of elements to the respective domain, i.e. to map sets of nodes $cs \subseteq F \cup D$ to the equivalent set $\{\gamma_x | x \in cs\}$, and vice-versa.

Definition 13 (Graph Correctness)**.** *Given are a system model S, a TFPG G, and an association map Γ. Also, let $EV \subseteq D(G) \cup F(G)$ and let $\Sigma(G)$ be the set of traces of $TS(G)$ that satisfy the Boolean constraints of the graph. We say that the graph of G is correct w.r.t. S, Γ and EV iff, for every discrepancy $d \in D(G) \cap EV$, the precedence constraints of d w.r.t. EV and $\Sigma(G)$ are equivalent, based on the mapping defined by Γ, to the precedence constraints of the corresponding expression γ_d w.r.t. $\{\gamma_v | v \in EV\}$ and S.*

This property guarantees an accurate graphical representation of the event orders of failure propagations possible in the system. In the running example, ignoring d_{crit}, we could obtain a complete TFPG by simply connecting f_{loc} to d_{stuck} and $d_{noncrit}$, and f_{cool} to $d_{noncrit}$. The graph however would not be correct, since in S, assuming the absence of f_{cool}, $d_{noncrit}$ is always preceded by both f_{loc} and d_{stuck}, which is not required by that graph.

This information on the ordering of events can be important in various situations related to fault protection design. For instance, for diagnosis, assume both f_{cool} and $d_{noncrit}$ are observable via some monitor, that f_{cool} is not activated and that $d_{noncrit}$ is activated. From this, based on the correct graph, we can deduce the occurrence of d_{stuck}, which is not possible in the other TFPG. The same information can also be important for choosing appropriate recovery strategies. Imagine that we have an efficient recovery procedure for when the robot is stuck but still in a nominal temperature range, and that recovery of a stuck robot with non-critical overheating is more complex and/or costly. The above information on precedence of events could then be used to justify a simpler recovery architecture.

Note that graph correctness is a property of just the graph of the TFPG and doesn't refer in any way to the edge labels. This means that we require the precedence constraints to be fully encoded by just the graph itself. Sometimes (but not always) these qualitative constraints could alternatively be encoded in the time bounds of edges. For instance, if two edges leave some common node, where the t_{min} value of the first one is higher than the t_{max} value of the second one, then we know that the activation of the target node of the second edge will always preceed the one of the target node of the first edge. It is however not clear what the advantage of such an encoding would be; a more selective representation of precedence constraints could be an interesting direction for future work.

We now formally define the TFPG synthesis problem.

Problem 1 (Graph Synthesis)**.** *Given are a system model S, a set of failure mode nodes F, a set of discrepancy nodes D, a set of modes M, and an association map Γ defining the nodes $x \in F \cup D$ w.r.t. S.* Graph Synthesis *consists in finding a TFPG G that satisfies the following properties: 1. $F = F(G)$ 2. $D \subseteq D(G)$ 3. G is complete w.r.t. S and Γ 4. the graph of G is correct w.r.t. S, Γ, and $EV = F \cup D$.*

Property 2 does not require D to be identical to $D(G)$, as it might be necessary to introduce additional nodes to model the precedence constraints, as described in the next sections.

3.5.1 Graph Synthesis

An algorithm to automatically synthesize TFPGs according to Problem 1 is now introduced. Algorithm 1 first instantiates all failure modes as `FM` nodes (Line 1). Next, all discrepancies given in input are defined as `OR` nodes (Line 2). At this point we still don't know whether some discrepancy effectively has `AND` node semantics, which will be inferred at a later point.

Algorithm 1 TfpgSynth

Inputs: system model S; set of failure modes F; set of discrepancies D; set of modes M; association map Γ.

Steps

```
 1: instantiate each failure mode $f \in F$ as an FM node;
 2: instantiate each discrepancy $d \in D$ as an OR node;
 3: for all $d \in D$ do
 4:     for all $mcs \in MCS(\gamma_d, \{\gamma_{d'} | d' \in F \cup D\}, S)$ do
 5:         instantiate a fresh virtual AND node $v$
 6:         create unconstrained edge $(v, d)$
 7:         for all $\gamma_{v'} \in mcs$ do
 8:             create unconstrained edge $(v', v)$
 9:         end for
10:     end for
11: end for
```

Next we iterate over the user-defined discrepancies (Line 3). For each such d we first compute all minimal cut-sets of the corresponding γ_d in the

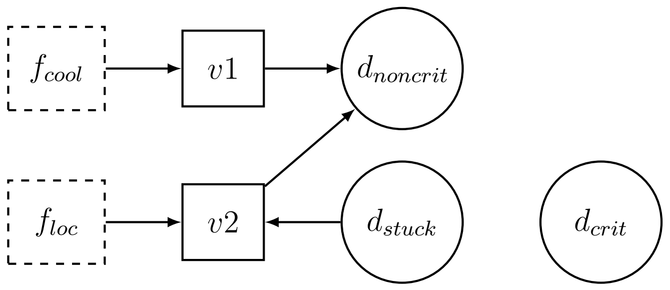


Figure 3.4: Intermediate result after iteration of Algorithm 1 for $d_{noncrit}$.

system (Line 4), which correspond to its precedence constraints. Since we don't have any a-priori knowledge about the possible sequences of events, we consider *all* other user-defined nodes in minimal cut-set computation. It is assumed that no discrepancy can have a minimal cut-set that doesn't contain a failure mode event, i.e. that discrepancies can occur independently of failure mode events. For each minimal cut-set mcs of γ_d a fresh virtual AND node is introduced (Line 5), and all nodes $v \in mcs$ are connected to it (Line 6-8). The activation of the virtual node represents the activation of all nodes in mcs, which itself enables the activation of the target node d. Every edge e at this point has maximally permissive constraints, i.e. with $tmin(e)$ set to 0, $tmax(e)$ set to $+\infty$, and $EM(e)$ set to M. This labeling ensures TFPG completeness by overapproximation. Figure 3.4 shows the intermediate result after the iteration for $d_{noncrit}$.

For the running example, at the end of the procedure we obtain the graph shown in Figure 3.5. The synthesized TFPG is complete w.r.t. S and Γ, as shown in Theorem 4.

Theorem 4. *A TFPG G built using Algorithm 1, based on a system model S and an association map Γ, is complete w.r.t. S and Γ.*

Proof. In a TFPG where $\forall e \in E$ we have $tmin(e) = 0$, $tmax(e) = +\infty$, and $EM(e) = M$, the proof obligations can be simplified. $\psi_{\texttt{OR}\cdot B}$ and $\psi_{\texttt{AND}\cdot B}$ trivially hold for all discrepancies since no delay value can be greater than

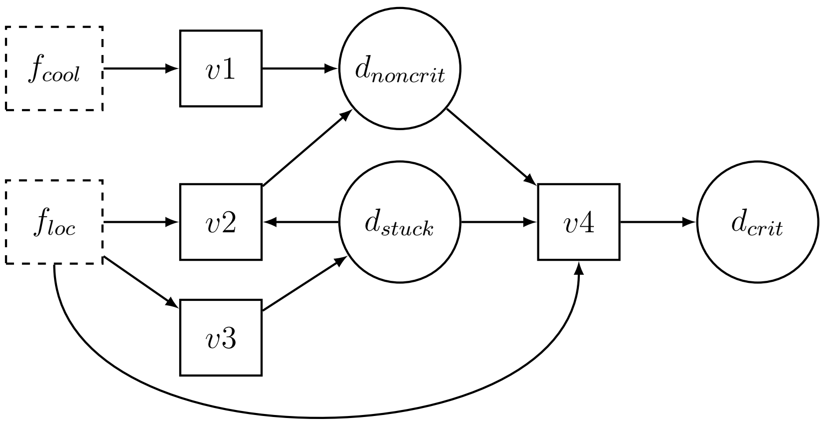


Figure 3.5: TFPG as produced by Algorithm 1 for running example, including user-defined and virtual nodes.

$+\infty$. $\psi_{\texttt{OR}\cdot A}$ can be simplified to $\mathsf{G}(\mathsf{O}\gamma_d \rightarrow \mathsf{O}(\mathsf{O}\gamma_d \wedge \bigvee_{(v,d)\in E}(\mathsf{O}\gamma_v)))$, and further to $\mathsf{G}(\mathsf{O}\gamma_d \rightarrow (\mathsf{O}\gamma_d \wedge \bigvee_{(v,d)\in E}(\mathsf{O}\gamma_v)))$. The $\mathsf{O}\gamma_d$ in the consequence can be dropped: $\mathsf{G}(\mathsf{O}\gamma_d \rightarrow \bigvee_{(v,d)\in E}(\mathsf{O}\gamma_v))$. Finally, since every AND node v is a virtual discrepancy, we obtain $\mathsf{G}(\mathsf{O}\gamma_d \rightarrow \bigvee_{(v,d)\in E}\bigwedge_{(v',v)\in E}(\mathsf{O}\gamma_{v'}))$, which is ensured by construction of the graph from the minimal cut-sets of γ_d. $\psi_{\texttt{AND}\cdot A}(d,\Gamma)$ can symmetrically be reduced to $\mathsf{G}(\mathsf{O}\gamma_d \rightarrow \bigwedge_{(v,d)\in E}(\mathsf{O}\gamma_v))$ and rewritten as $\mathsf{G}(\bigwedge_{(v,d)\in E}(\mathsf{O}\gamma_v) \rightarrow \bigwedge_{(v,d)\in E}(\mathsf{O}\gamma_v))$, which trivially holds. □

In addition to completeness, the graph of the synthesized TFPG also satisfies the correctness property. Note that we are only interested in graph correctness w.r.t. the user-defined nodes.

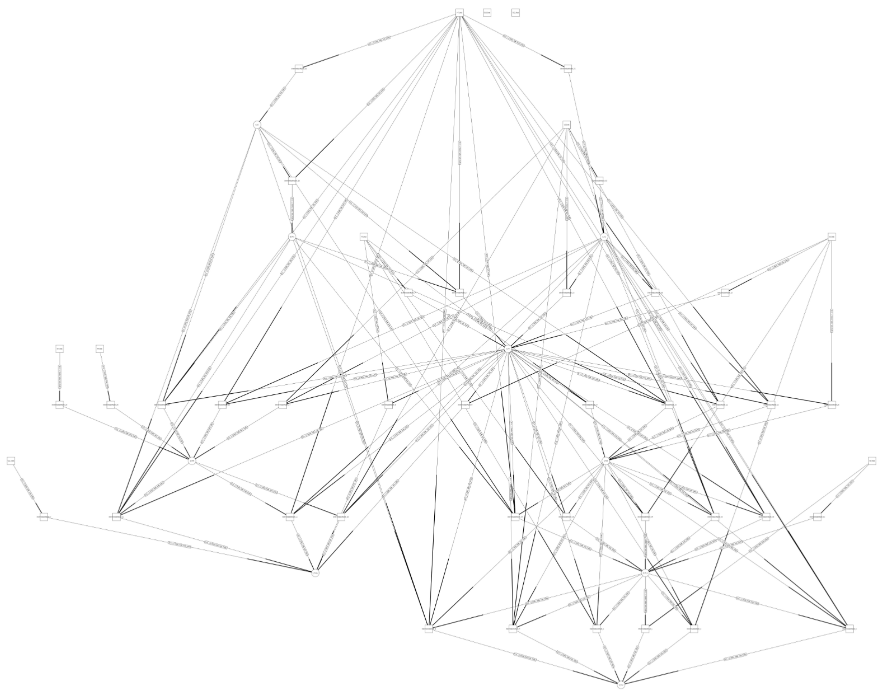
Theorem 5. *Given are a system model S, an association map Γ, and a TFPG G as produced by Algorithm 1 for failure mode nodes F and discrepancies D. As per Definition 13, the graph of G is correct w.r.t. S, Γ, and $EV = F \cup D$.*

Proof. For every discrepancy $d \in D$ the following holds. The set of all sets of nodes $cs \subset EV$ connected to d via some respective virtual AND node v (i.e., where $(v,d) \in E$, and $v' \in cs$ iff $(v',v) \in E$) represents by construction, when mapped to S, the set of all minimal cut-sets of γ_d. It also represents all minimal cut-sets of d w.r.t. $\Sigma(G)$. Otherwise, one such set of nodes cs would exist that is not a minimal cut-set for d in $\Sigma(G)$. If cs is a cut-set, it is automatically minimal by the semantics of the nodes. If it is not a cut-set, then for at least one $d' \in D \cap (cs \cup d)$ there are not sufficient nodes in $(cs \cup d)$ such that one virtual `AND` node v, with $(v, d') \in E$, can be activated. This however is not possible, since from the FTA step we know that $\{\gamma_v | v \in cs \cup d\}$ contains a minimal cut-set for every $\gamma_{d'}$, with $d' \in D \cap (cs \cup d)$. $\square$

3.5.2 Graph Simplification

Even though a TFPG synthesized by the procedure above satisfies certain relevant properties, its graph structure might be too verbose for some applications. For instance, manual inspection by a safety engineer is often impractical with the full graph and requires a simpler version that still maintains completeness and correctness. Figure 3.6 compares the full graph for the model Cassini2 described in Section 3.7, and the one simplified with the procedure described in this section. For an engineer it will be very difficult to understand the propagation behavior from the fully verbose graph in Figure 3.6a, but relatively easy with the one in Figure 3.6b. Even in the running example it is not straight-forward to manually reconstruct the propagation patterns from the synthesis result in Figure 3.5, as compared to the TFPG in Figure 3.1. Some simplification however is possible: the edge $(f_{loc}, v2)$ is redundant, and all virtual `AND` nodes except $v4$ are also not essential to encode the precedence constraints among the user-defined nodes.

For the purpose of TFPG simplification we introduce the following theorem, based on which a procedure to remove edges while preserving completeness and correctness can be derived. It uses the formula $\phi_{prec}(G) :=$


(a) without simplification (56 nodes, 141 edges)

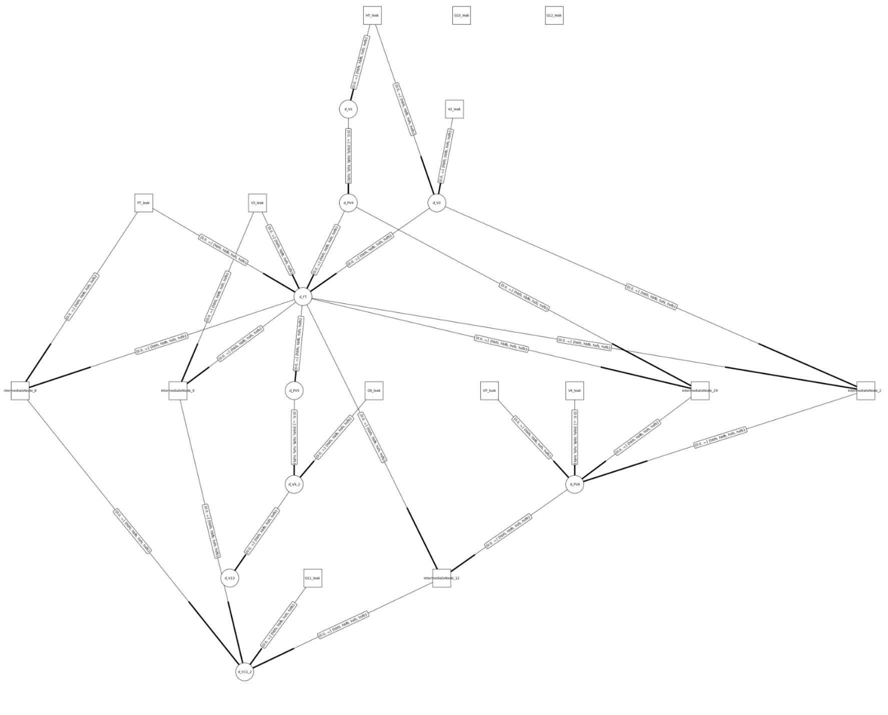

(b) with simplification (24 nodes, 30 edges)

Figure 3.6: Effect of simplification procedure on Cassini2 model (rendered with xSAP).

$\bigwedge_{d \in D}(\mathbf{d} \rightarrow \bigvee_{(v,d) \in E(G)} \bigwedge_{(v',v) \in E(G)} \mathbf{v}')$, which encodes the Boolean constraints of G, similarly to $\phi_{bool}(G)$, but factors out the virtual `AND` nodes, as the correctness property only regards the user-defined nodes. We write $\overline{cs}$ for the valuations of $\phi_{prec}(G)$ and $\phi_{bool}(G)$ that assign $\top$ to any $v \in cs$, and $\bot$ to all other variables.

Theorem 6. *Given is a TFPG G as produced by Algorithm 1 for failure mode nodes F and discrepancies D, map Γ and system model S. Also, given is a second TFPG G' that has the same nodes as G, the same edges towards* `OR` *nodes, but a subset of the edges towards the* `AND` *nodes. Then, G' is complete w.r.t. Γ and S; also, if $\phi_{prec}(G) \equiv \phi_{prec}(G')$, the graph of G' is correct w.r.t. Γ, S, and $EV = F \cup D$.*

Proof. Due to maximally permissive edge constraints and the expressions associated to virtual discrepancies, we can ignore the proof obligations $\psi_{\texttt{OR}\cdot B}$, $\psi_{\texttt{AND}\cdot A}$, and $\psi_{\texttt{AND}\cdot B}$, as shown in the proof of Theorem 4. Removing edges towards virtual `AND` nodes effectively weakens the proof obligations $\psi_{\texttt{OR}\cdot A}$, thus completeness is preserved.

Then, for both G and G' we have that $cs \subset F \cup D$ is a cut-set of $d \in D$ in $\Sigma(G)$ *iff* $\overline{cs \cup d} \models \phi_{prec}(G)$. ($\Rightarrow$) Assume cs is a cut-set of d in $\Sigma(G)$; then there exists $V \subseteq \texttt{AND}(G)$ such that $\overline{V \cup cs \cup d}$ is reachable on some $\pi \in \Sigma(G)$, and hence $\overline{cs \cup d} \models \phi_{prec}(G)$, since $\phi_{bool}(G) \rightarrow \phi_{prec}(G)$. ($\Leftarrow$) Assume $\overline{cs \cup d} \models \phi_{prec}(G)$, then $\overline{V \cup cs \cup d} \models \phi_{bool}(G)$, with $V \subseteq \texttt{AND}(G)$ and $v \in V$ iff $\overline{cs \cup d} \models \bigwedge_{(v',v) \in E(G)} v'$, of which there is at least one for every $d' \in D \cap (cs \cup d)$ as guaranteed by $\phi_{prec}(G)$, from which it follows that $\overline{V \cup cs \cup d}$ is reachable on some $\pi \in \Sigma(G)$ at $\pi[0]$, as it is a state of $TS(G)$ that satisfies $\phi_{bool}(G)$, thus qualifying as an initial state of a trace $\pi \in \Sigma(G)$, and hence cs is a cut-set of d in $\Sigma(G)$.

Finally, from $\phi_{prec}(G) \equiv \phi_{prec}(G')$ it follows that every $d \in D$ has the same cut-sets in $\Sigma(G)$ and $\Sigma(G')$ w.r.t. EV, and thus also the same mini-

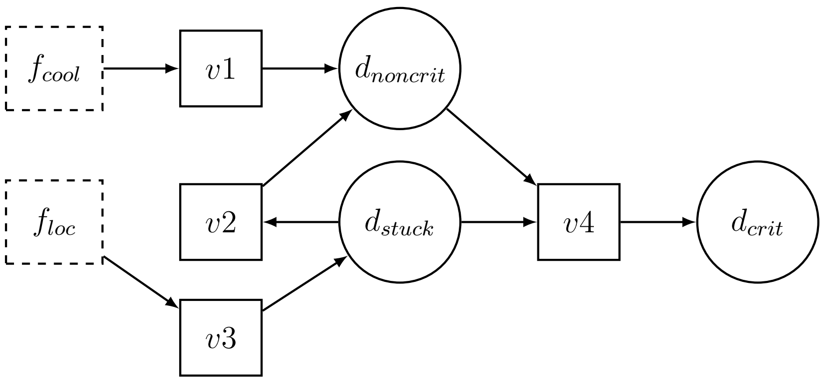


Figure 3.7: Result of removing redundant edges from synthesis result shown in Figure 3.5.

mal cut-sets. □

Note that Theorem 6 does not allow to remove *all* edges towards any `AND` node, as this would result in an illegal TFPG. The only nodes that can (must) have no incoming edge are failure mode nodes.

Given a TFPG G that was computed by Algorithm 1, we start to drop edges towards the virtual `AND` nodes, resulting in new TFPGs G'. The theorem then tells us that completeness and graph correctness are not affected as long as $\phi_{prec}(G)$ and $\phi_{prec}(G')$ are equivalent, which can be checked for instance with a SAT solver. We thus remove as many edges as possible until the removal of any further edge would either result in an illegal TFPG or break graph correctness. For the running example, the result of removing these edges is shown in Figure 3.7.

After removing redundant edges of G, guided by $\phi_{prec}(G)$, it might also be possible to eliminate some virtual `AND` nodes without affecting the completeness of G and the correctness of its graph. A virtual `AND` node v with a single incoming edge (v', v) and a single outgoing edge (v, d), e.g. $v1$ in Figure 3.7, can be removed by replacing the two edges involving v with a new edge (v', d). The result of this step is shown in Figure 3.8.

Furthermore, if two `AND` nodes exist that have identical incoming and

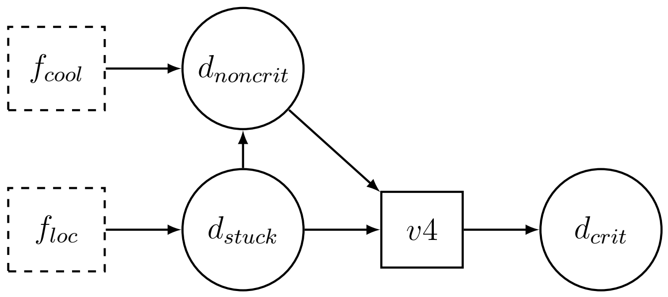


Figure 3.8: Result of removing unnecessary `AND` nodes from the TFPG in Figure 3.7.

outgoing edges, one of them can be simply dropped. An `OR` node d with a single incoming edge from a virtual `AND` node v can be redeclared as an `AND` node; then for every edge (v', v) a new edge (v', d) is introduced and v is dropped from G; in Figure 3.8 this applies to d_{crit}. By applying all simplification strategies to the TFPG in Figure 3.5 we finally obtain the one in Figure 3.1.

3.5.3 Edge Tightening

At this point, the completeness checks presented in Section 3.4 can be used to tighten the edges of the TFPG according to Definition 10, in order to identify more precise constraints on propagation delays and contexts. Note that edge tightening can be performed only after simplification, as the latter ignores propagation delays and assumes maximally permissive edges. In future work we will investigate the need and corresponding methods for a closer integration between graph simplification and tightening, for instance to enable a more fine-grained modeling of propagation delays.

3.6 Implementation

The prototype created for the present work is implemented on top of the safety analysis platform xSAP [Bittner et al., 2016a], which in turn is based on nuXmv [Cavada et al., 2014], a symbolic model checker for infinite state transition systems modeled in the SMV language. The implementation, containing all algorithms described in this chapter, can be downloaded at `http://xsap.fbk.eu`. In the following we describe the implementation of the techniques for behavioral validation (Section 3.4) and synthesis (Section 3.5) of TFPGs.

Completeness Check

At the core of the implementation we use a reduction of the completeness check via MTL proof obligations to a reachability problem. This allows us to directly reuse the reachability algorithms of nuXmv off-the-shelf for behavioral validation – as well as the parameter synthesis techniques described in Section 4.4 for automated tightening of the edges.

Intuitively, the reduction works as follows. For expressions of the type $\mathsf{O}\gamma_d$ we extend the system model with corresponding history monitors. Furthermore we introduce one timer per edge e that measures the duration for which the corresponding expression $(\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)}$ has been true on the current path; the timer is disabled when the edge is not active and frozen when the target discrepancy is activated. An example of the corresponding SMV code for the running example is shown in Figure 3.9. Note that we assume the presence of an input variable that is equal to the difference in timestamps between adjacent states – variable `t_#delta` in Figure 3.9. For the purpose of our analysis this makes it also unnecessary to keep track of the absolute time stamps.

Based on this, the MTL proof obligations are expressed as invariance proof obligations, which can be solved with standard reachability algorithms. For instance, to verify the proof obligation $\psi_{\mathtt{OR}\cdot A}(d, \Gamma)$, we check whether the timer value of at least one edge reaching the respective `OR` discrepancy is greater or equal to the corresponding t_{min} value when the discrepancy is activated.

```
DEFINE EState_EDGE2 := failuremode_FailCooling & Mode_CriticalZone;
VAR ETime_EDGE2 : real;

ASSIGN init(ETime_EDGE2) := case
  EState_EDGE2 : 0;
  TRUE : -1;
  esac;

ASSIGN next(ETime_EDGE2) := case
  or_node_NonCriticalOverheating : ETime_EDGE2;
  !next(EState_EDGE2) : -1;
  !EState_EDGE2 & next(EState_EDGE2) : 0;
  TRUE : ETime_EDGE2 + t_#delta_0;
  esac;
```

Figure 3.9: SMV code for the timer of edge $(f_{cool}, d_{noncrit})$ in ForgeRobot.

The proof of the following theorem shows in detail how this reduction works.

Theorem 7. *Given are a system model S, a trace π of S, an association map Γ relating S to a given TFPG G, and an* `OR` *discrepancy d of G. The proof obligations $\psi_{\texttt{OR}\cdot A}(d,\Gamma)$ and $\psi_{\texttt{OR}\cdot B}(d,\Gamma)$ of Theorem 1 can be equivalently expressed as invariant properties $\psi_{\texttt{OR}\cdot A\cdot invar}(d,\Gamma)$ and $\psi_{\texttt{OR}\cdot B\cdot invar}(d,\Gamma)$, such that:*

- $\pi \models \psi_{\texttt{OR}\cdot A}(d,\Gamma)$ iff $\forall k \in \mathbb{N} \cdot \pi[k] \models \psi_{\texttt{OR}\cdot A\cdot invar}(d,\Gamma)$;
- $\pi \models \psi_{\texttt{OR}\cdot B}(d,\Gamma)$ iff $\forall k \in \mathbb{N} \cdot \pi[k] \models \psi_{\texttt{OR}\cdot B\cdot invar}(d,\Gamma)$.

Likewise, for `AND` *nodes properties the proof obligations $\psi_{\texttt{AND}\cdot A}(d,\Gamma)$ and $\psi_{\texttt{AND}\cdot B}(d,\Gamma)$ of Theorem 2 can be equivalently expressed as invariant properties $\psi_{\texttt{AND}\cdot A\cdot invar}(d,\Gamma)$ and $\psi_{\texttt{AND}\cdot B\cdot invar}(d,\Gamma)$, such that:*

- $\pi \models \psi_{\texttt{AND}\cdot A}(d,\Gamma)$ iff $\forall k \in \mathbb{N} \cdot \pi[k] \models \psi_{\texttt{AND}\cdot A\cdot invar}(d,\Gamma)$;

- $\pi \models \psi_{\texttt{AND}\cdot B}(d, \Gamma)$ iff $\forall k \in \mathbb{N} \cdot \pi[k] \models \psi_{\texttt{AND}\cdot B\cdot invar}(d, \Gamma)$.

Proof. For every node $v \in \{F \cup D\}$ we introduce a fresh Boolean variable h_v with $\pi[k] \models h_v \leftrightarrow \mathsf{O}\gamma_v$ for every state k of π. For every edge $e = (v, d) \in E$ we introduce a fresh Boolean variable $estate_e$ with $\pi[k] \models estate_e \leftrightarrow ((\mathsf{O}\gamma_v) \wedge \gamma_{\mu(e)})$ for every state k of π. $\psi_{\texttt{OR}\cdot A}(d, \Gamma)$ can then be rewritten as follows:

- $\mathsf{G}(h_d \to \mathsf{O}(h_d \wedge \bigvee_{e=(v,d)\in E}(estate_e \mathsf{S}^{\geq tmin(e)} estate_e)))$
- *iff*, using timers $etime_e$ that are reset to 0 when the respective $estate_e$ changes truth value:
  $\mathsf{G}(h_d \to \mathsf{O}(h_d \wedge \bigvee_{e=(v,d)\in E}(estate_e \wedge etime_e \geq tmin(e))))$
- *iff*, by disabling the timers $etime_e$ (setting them to -1) when the respective $estate_e$ evaluates to false and given that $tmin(e) \geq 0$:
  $\mathsf{G}(h_d \to \mathsf{O}(h_d \wedge \bigvee_{e=(v,d)\in E}(etime_e \geq tmin(e))))$
- *iff*, by freezing $etime_e$ as soon as h_d becomes true:
  $\mathsf{G}(h_d \to (h_d \wedge \bigvee_{e=(v,d)\in E}(etime_e \geq tmin(e))))$
- *iff*, by simplification of tautology:
  $\mathsf{G}(h_d \to \bigvee_{e=(v,d)\in E}(etime_e \geq tmin(e)))$

The first step substituting the S-expression can be done since, for all k and any edge e we have $\pi[k] \models (estate_e \mathsf{S}^{\geq tmin(e)} estate_e)$ iff $\pi[k] \models (estate_e \wedge etime_e \geq tmin(e))$, assuming a timer $etime_e$ that is reset to 0 when the corresponding $estate_e$ changes truth value, and else increments by the difference in timestamps. This can be shown considering the following four cases:

- for $k = 0$, consider two subcases: for $tmin(e) = 0$, the truth value of both expressions depends on the truth value of $estate_e$, and for $tmin(e) > 0$, both expressions always evaluate to false;

- if $\pi[k] \models \neg estate_e$, both expressions evaluate to false;
- if $\forall i \leq k \cdot \pi[i] \models estate_e$, it also holds since $\tau_k = etime_e(\pi[k])$, and the truth value of both expressions depends on whether $\tau_k >= tmin(e)$;
- the case remains where $k > 0$, $\pi[k] \models estate_e$, and $\exists i < k \cdot \pi[i] \models \neg estate_e$, from which it follows that $\exists i < i' \leq k$ s.t. $\pi[i'-1] \models \neg estate_e$, $\pi[i'] \models estate_e$, and $\forall i' \leq j \leq k \cdot \pi[j] \models estate_e$, and therefore $\tau_k - \tau_{i'} = etime_e(\pi[k])$; both expressions thus depend on whether $\tau_k - \tau_{i'} \geq tmin(e)$

The second step that drops the $estate_e$ variable is possible, since in states where $estate_e$ is false, the expressions specific to e evaluate both to false, and in the other cases the timers have the same value.

The once operator can be dropped during the third step, assuming that $etime_e$ freezes when h_d becomes true, since:

- $\mathsf{G}((h_d \wedge \bigvee_{e=(v,d)\in E} etime_e \geq tmin(e)) \leftrightarrow$
  $\mathsf{G}(h_d \wedge \bigvee_{e=(v,d)\in E} etime_e \geq tmin(e)))$
- and hence
  $\mathsf{G}(\mathsf{O}(h_d \wedge \bigvee_{e=(v,d)\in E} etime_e \geq tmin(e)) \leftrightarrow$
  $(h_d \wedge \bigvee_{e=(v,d)\in E} etime_e \geq tmin(e)))$

Then, let $\psi_{\mathtt{OR}\cdot A\cdot invar}(d,\Gamma) := h_d \rightarrow \bigvee_{e=(v,d)\in E}(etime_e \geq tmin(e))$.
Similarly, for $\psi_{\mathtt{OR}\cdot B}(d,\Gamma)$ we have the following reduction.

- $\mathsf{G}\neg(\bigvee_{e=(v,d)\in E}(estate_e \wedge \neg h_d)\mathsf{S}^{>tmax(e)}(estate_e \wedge \neg h_d))$
- *iff*, since we have $\pi \models \mathsf{G}(\neg h_d \rightarrow \mathsf{H}(\neg h_d))$ and by Boolean equivalence,
  $\mathsf{G}\neg(\neg h_d \wedge \bigvee_{e=(v,d)\in E}(estate_e \mathsf{S}^{>tmax(e)} estate_e))$
- *iff*, using timers $etime_e$ that are reset to 0 when the respective $estate_e$ changes truth value:
  $\mathsf{G}\neg(\neg h_d \wedge \bigvee_{e=(v,d)\in E}(estate_e \wedge etime_e > tmax(e)))$

- *iff*, by disabling the timers $etime_e$ (setting them to -1) when the respective $estate_e$ evaluates to false and given that $tmax(e) \geq 0$:
  $\mathsf{G}\neg(\neg h_d \land \bigvee_{e=(v,d)\in E} etime_e > tmax(e))$
- *iff*, by freezing $etime_e$ as soon as h_d becomes true:
  $\mathsf{G}\neg(\neg h_d \land \bigvee_{e=(v,d)\in E} etime_e > tmax(e))$

The last two derivations are performed such that we can reuse in the implementation the same timers $etime_e$ as for the proof obligations $\psi_{\texttt{OR}\cdot A}$. The last equivalence holds, because for all $k \in \mathbb{N}$ the invariants in both properties hold when $\pi[k] \models h_d$, and otherwise each pair of associated timers has the same value and thus both invariants hold for any given k, or they both don't hold. Then, let $\psi_{\texttt{OR}\cdot B\cdot invar}(d,\Gamma) := \neg(\neg h_d \land \bigvee_{e=(v,d)\in E} etime_e > tmax(e))$.

The derivation of $\psi_{\texttt{AND}\cdot A\cdot invar}(d,\Gamma)$ and $\psi_{\texttt{AND}\cdot B\cdot invar}(d,\Gamma)$ is analogous, except that "$\bigwedge_{e=(v,d)\in E}$" is used instead of "$\bigvee_{e=(v,d)\in E}$". Then, let
$\psi_{\texttt{AND}\cdot A\cdot invar}(d,\Gamma) := h_d \to \bigwedge_{e=(v,d)\in E}(etime_e \geq tmin(e))$, and
$\psi_{\texttt{AND}\cdot B\cdot invar}(d,\Gamma) := \neg(\neg h_d \land \bigwedge_{e=(v,d)\in E} etime_e > tmax(e))$.

$\square$

We remark that the reachability problem for infinite-state transition systems is in general undecidable, and plan to adress the computational complexity of decidable subclasses in future work. The fact that the proof obligations require only a very restricted subset of MTL might be an advantage in this regard.

Tightness Check and Automated Tightening

For checking tightness and for automated tightening, we assume that we are working with a TFPG that is complete and that every discrepancy is reachable on some system trace; else, checking and improving tightness (on certain edges) doesn't make sense to begin with.

For mode labels, we try do drop individual modes and check whether the resulting TFPG is still complete; for tightening we simply drop as many modes as possible. For t_{min} bounds and for t_{max} bounds set to a value in $\mathbb{N}$, we perform a search in discrete steps, according to the highest precision of any time constant in the original TFPG. Indeed, in practice the precision of interest is always finite. Tightness is thus guaranteed w.r.t. the chosen precision and the resulting discretized search domain. The idea is then to explore the resulting lattice of solutions top-down; for the simple tightness check we just perform one step, for full tightening we proceed until the parameter under consideration cannot be tightened anymore. With respect to tightening, the advantage of this top-down approach is that we obtain an anytime-algorithm for refinement; it can be interrupted any time, and the result will still be a complete TFPG.

Note that for finite t_{max} values, the search domain for tightening is finite, whereas for t_{min} it is finite only if the corresponding t_{max} is an element of $\mathbb{N}$, and is infinite if $t_{max} = \infty$. Nevertheless, the amount of values that need to be considered for such t_{min} parameters is finite, because all discrepancies are assumed to be reachable in the system model – hence a lower bound for the propagation delay exists.

For an edge $e = (v, d)$ with $tmax(e) = +\infty$ the tightness and tightening problems effectively ask whether the TFPG would still be complete by setting $tmax(e)$ to some finite value, and possibly to identify this value. This is a difficult problem, because the search domain is indeed infinite. Proving techniques going beyond the ones we use here would be necessary for completeness in showing that no value in $\mathbb{N}$ is a good assignment for the t_{max} under consideration.

For tightness, an easier but incomplete check consists in finding a trace where the edge e is always active, time diverges, but d never activates – assuming d is an `OR` node; this proves that no finite value for t_{max} exists, because even though time advances, the propagation never occurs. The implementation uses this approach. For automated tightening instead we limit the search for the t_{max} value by an upper bound; if no valid choice for t_{max} is found within this range, we set it to ∞.

Depending on the application at hand, this might also be sufficient for practical purposes. This applies for instance to the use of TFPGs as a basis to design an FDIR architecture, the central objective of the applications discussed in Chapter 5. In such a context, accurate upper propagation bounds are mostly relevant for propagation towards monitors, and a monitor that is not triggered within a certain amount of time is not a useful one – indeed the motivation is similar to the one for BoundDel specifications in diagnosability analysis. Furthermore, for propagations towards discrepancies representing failures or feared events, the t_{min} value is the one that actually drives the design of recovery strategies, representing the worst case requirement on response time.

Graph Synthesis

For graph synthesis we use on the procedures of [Bozzano et al., 2015d] for minimal cut-set computation off-the-shelf, as implemented in xSAP. The algorithm presented in Section 3.5.1 is then used to instantiate the TFPG with maximally permissive edges.

Graph Simplification

The graph simplification step was implemented based on the SMT solver MathSAT [Cimatti et al., 2013b]. It uses two copies of $\phi_{prec}(G)$, one of which is parameterized on all edges towards `AND` nodes. More precisely, we introduce one Boolean variable $p_{(v',v)}$ for each edge (v', v) that could potentially be removed, and modify $\phi_{prec}(G)$ as follows:

$$\phi_{prec-param}(G) := \bigwedge_{d \in D} (\mathbf{d} \rightarrow \bigvee_{(v,d) \in E(G)} \bigwedge_{(v',v) \in E(G)} (\mathbf{p}_{(\mathbf{v'},\mathbf{v})} \rightarrow \mathbf{v'})$$

Setting such a parameter to *true* forces the respective node v' to be active as in the original formula. Setting it to *false* makes the implication automatically *true*, and whether v' is active or not doesn't influence the satisfaction of the disjunction. The overall formula is thus logically equivalent to $\phi_{prec}(G')$, where G' is the result of dropping the edges from G according to parameters $p_{(v',v)}$ set to *false*.

We then iteratively check the satisfiability of the formula:

$$\phi_{simplify}(G) := \neg\phi_{prec}(G) \wedge \phi_{prec-param}(G)$$

A model satisfying this formula represents an activation pattern possible in the simplified TFPG, but not in the original one; the other way around is not possible as by removing edges we monotonically increase the legal activation patterns. With all parameters set to true, i.e. with all edges enabled, the formula $\phi_{simplify}(G)$ will be unsatisfiable, which means that both graphs accept exactly the same patterns. We then iteratively remove edges until the parameters set to true represent a minimal unsatisfiable core, i.e. removing any of the edges still enabled would introduce new activation patterns over the user-defined nodes.

Checking Graph Correctness

Checking graph correctness for a given TFPG means to compare the precedence constraints of every discrepancy d in G and the associated precedence constraints of γ_d in the system model S. Thus we need to compute all minimal cut-sets of γ_d, as done in the synthesis algorithm, and compare them with the minimal $\top$ assignments to $d \wedge \phi_{bool}$. For our simple implementation we rely on a SAT solver and a modified all-SAT procedure which prunes models from the search space that have been proven not to be minimal.

3.7 Experimental Evaluation

In this section we provide an experimental evaluation of the developed algorithms for completeness checking, automated tightening of the edges, and graph synthesis and simplification. In the benchmarks we are primarily interested of all in the feasibility of the proposed algorithms, considering a variety of use cases. The results show that the algorithms do indeed scale up to interesting problem sizes. Also the effect of certain TFPG properties on runtime is studied. Comparative benchmarks cannot be performed, as the specific problems are introduced in the present work for the first time.

We first present the use cases for which we perform various benchmarks and the testing infrastructure. Then a first set of experiments is described for the analyses that work with a given TFPG, i.e. behavioral validation and edge tightening. Finally we describe the experiments done for TFPG synthesis.

3.7.1 Use Cases and Set-Up

We use the following use cases. ACEX and AUTOGEN are hand-crafted models based on a state space derived from partially random graphs, containing discrete clocks. BATTERY SENSOR describes a timed generator-battery system that powers a hypothetical device. It is also mainly discrete with real-valued clocks. CASSINI are variants of the spacecraft propulsion system described in [Williams and Nayak, 1996], enriched with timed aspects. It is composed of two engines fed by redundant propellant/gas circuit lines, which contain several valves and pyro-valves. GUIDANCE is a discrete model of the Space Shuttle engines contingency guidance requirements. FORGEROBOT are variations of the running example. POWERDIST describes the fault protection logic of a power distribution system consisting of circuit breakers, switches, and power lines. Additionally we ran our implementation on two discrete untimed industrial models, WBS [Bozzano et al., 2015a] describing an aircraft wheel-braking system, and X34 [Bajwa and Sweet, 2003] describing the propulsion system of an experimental spacecraft.

model	max. bool	max. real	max. FM	max. D	max. E
acex	35	0	2	25	26
autogen	99	0	8	20	29
battery	43	5	4	9	14
cassini	301	10	16	16	38
forge	25	7	6	13	23
guidance	98	0	6	9	19
pdist	84	0	7	7	21
wbs	1179	0	12	11	19
x34	553	0	9	18	32

Table 3.1: Use-case statistics with maximum number of Boolean and real variables of respective system models, maximum numbers of user-defined failure modes, discrepancies, and edges.

For each of the 9 use cases we have one system model encoded in SMV. For ACEX, CASSINI, and FORGEROBOT we also created parametric variations of the models, resulting in a total of 14 different system models used in the benchmarks. Figure 3.1 gives, for each use case, the number of variables in the model, the maximum number of user-defined failure modes and discrepancies, as well as the maximum number of edges for the TFPGs used in behavioral validation and edge tightening.

All tests described in the following were run on a dedicated 64bit Linux computer with a 12 core CPU at 2.67 GHz and 100GB of RAM. 4 cores were reserved for each test run to limit time skew due to parallel executions of the tests. Each test was executed on a single core with a time limit of 3600 seconds and a memory limit of 4GB.

All benchmarks used in this chapter are available at `http://es.fbk.eu/people/bittner/phd_tfpg_expeval.tar.bz2`.

3.7.2 Behavioral Validation and Automated Tightening

In the first set of benchmarks we want to study the performance of techniques working with a given TFPG, that is, one where nodes, edges, and edge labels are given. In total we created 72 tight TFPGs by taking a default TFPG for each system model and creating new instances by incrementally removing discrepancies from it. For all of these instances except for the untimed WBS and X34, we created two "relaxed" versions of the TFPGs by decreasing or increasing individual edge parameters, resulting in a overall number of 212 use cases. The instances thus reflect various degrees in graph complexity and various "degrees" of edge tightness.

For each instance we ran the completeness check and the tightening procedure. We didn't run the tightness checks, as their runtime strongly depends on the ordering in which parameters are checked, and how many iterations over tight assignments have to be performed before finding the first non-tight assignment. However, bounds for the runtime can be derived from the completeness check on the lower end, and the tightening procedure on the upper end. Tightness checking is similar to tightening: it also starts with a completeness check, but then stops at the first non-tight assignment. Note that for the 72 instances that are already tight, the tightness check has the same runtime behavior as the automated tightening procedure.

All completeness checks terminated within the timeout (all except four within 800s) and IC3 was able to prove completeness in all cases. Furthermore, the check for WBS terminated after 67s, and the one for X34 after 21s. Also most tightening runs terminated within the timeout bound, except 10 that went out-of-time and 5 that went out-of-memory. Table 3.2 summarizes the results, which show the feasibility of the approach, also

model	instances	completeness		tightening			
		μ	σ	μ	σ	o.o.t.	o.o.m.
acex-10	15	171	71	731	512	0	0
acex-12	33	334	106	838	321	1	3
autogen	66	156	147	925	761	0	0
battery	12	23	29	71	25	0	0
cassini-2	15	28	19	75	69	0	0
cassini-4	39	322	248	1179	973	3	0
forgerobot1	3	103	56	160	30	0	0
forgerobot2	3	2	0	10	2	0	0
forgerobot3	3	24	3	224	66	0	0
forgerobot4	3	145	16	N.A.	N.A.	1	2
guidance	12	14	8	94	72	0	0
pdist	6	622	197	2776	0	5	0
wbs	1	67	0	N.A.	N.A.	0	0
x34	1	21	0	N.A.	N.A.	0	0

Table 3.2: Completeness and tightening runtimes in seconds per system model (mean and standard deviation for all instances).

for automated tightening, even though as expected it is more difficult than completeness checking.

These results are encouraging, because the use cases are representative for unit to subsystem-level complexity of safety analyses, in terms of the system model and in terms of the amount of basic failure modes and effects. This clearly shows the feasibility in a real project setting, a finding which is also confirmed by the case studies described in Chapter 5.

When comparing the runtimes at different levels of relaxation (tight vs. relaxed t_{min} bounds vs. relaxed t_{min} and relaxed t_{max} bounds), there are no significant differences for the completeness check. However, for the tightening procedure the "tightness degree" of the input TFPG does matter, as can be seen in Figure 3.10. This shows by giving providing a first reasonably accurate approximation of the parameters, an engineer

num. edges	completeness	tightening
3	2	4
8	14	96
18	134	681
23	246	1030
27	265	1473
31	172	1822
35	279	2355
38	261	3086

(a) incremental topology

num. edges	completeness	tightening
30	621	2836
31	914	*n.a.*
32	401	*n.a.*
34	322	3513
36	457	2777
38	261	3086

(b) refined topology

Table 3.3: Effect on completeness/tightening runtimes (in seconds) of increasing TFPG complexity.

could speed up the fine-tuning by the automated tightening procedure.

Finally, in instances that were generated by merely adding new nodes and edges to the existing graph (e.g. given some edge $A \rightarrow B$, adding a new node C and an edge $B \rightarrow C$), we can notice for completeness checking and for tightening a corresponding increase in runtime. However, when we increase the TFPGs complexity by refining existing edges (e.g. replacing an edge $A \rightarrow C$ with a new node and corresponding pairs of edges $A \rightarrow B \rightarrow C$) instead of simply extending the graph, a reliable prediction of the corresponding effect on runtime is not possible. Table 3.3 shows these findings for the CASSINI-4 model and the corresponding TFPGs with relaxed edge parameters. In a project it can be useful to have an estimate on the runtime for a run of completeness verification and/or automated tightening, and these results show in what cases this could be done and in what cases most likely not.

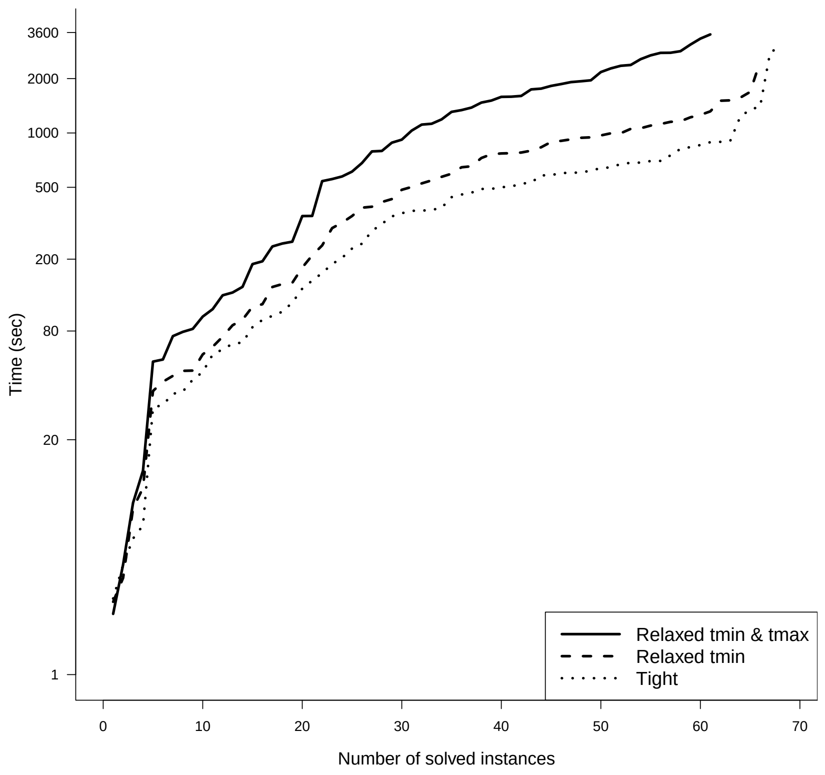
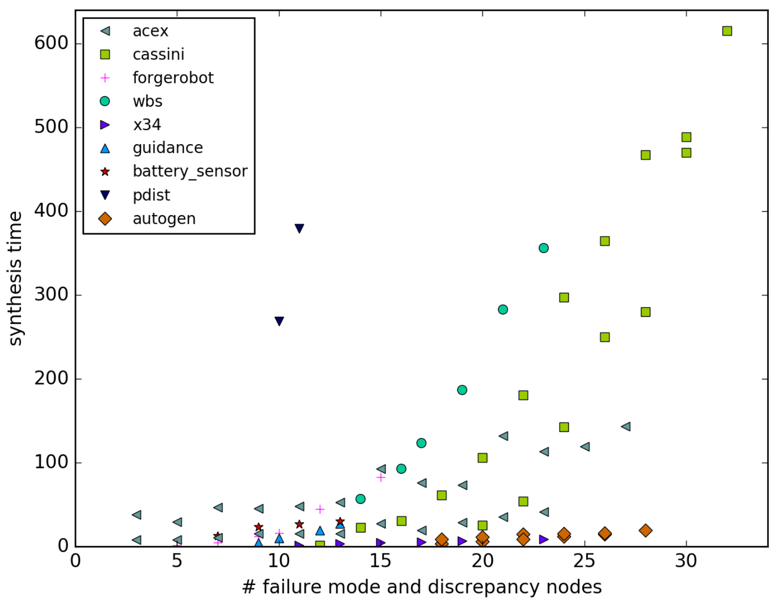


Figure 3.10: Tightening difficulty (time in seconds) for each class of relaxed TFPGs.

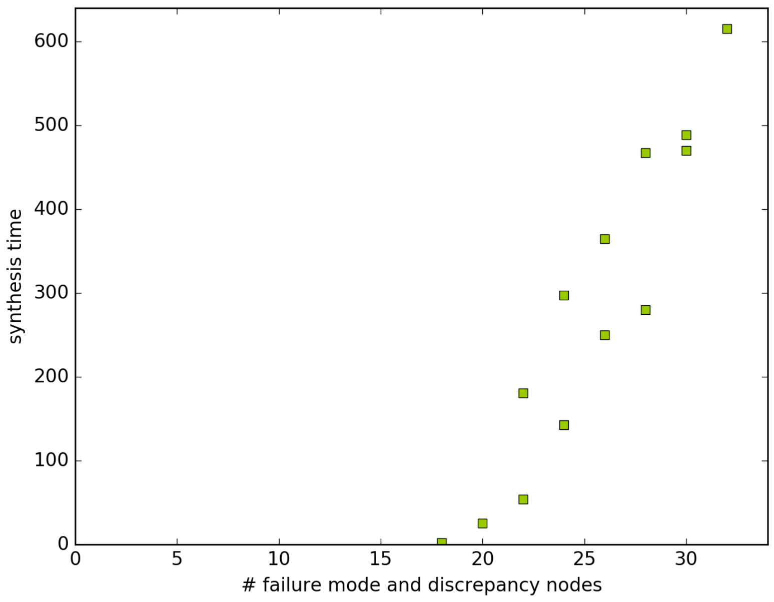
3.7.3 Graph Synthesis

In the second set of benchmarks we study the performance of the synthesis and simplification algorithms. As the input for this problem is only a system model, a set of nodes, and an association map, we created another set of test instances. Specifically, we took as inputs the nodes of all tight TFPGs used in Section 3.7.2 and created further instances for the use cases WBS and X34, resulting in a total of 82 instances.

For each instance we ran the synthesis and simplification procedures. All benchmarks terminated well within the timeout. Also on the two industrial models we obtained good results: X34 turned out to have a maximum run-time of 9s, and WBS of 355s. These results show the feasibility of the approach, since, on models with a complexity comparable to unit-to-



(a) all instances



(b) Cassini4 instances

Figure 3.11: Number of nodes vs. synthesis time (in seconds).

subsystem-level problems, it terminates in reasonable time.

The time needed for the simplification step is almost instantaneous in all cases, always below 0.3 seconds. The dominant component of the overall runtime is thus the computation of minimal cut-sets. However, the effect of simplification is considerable: it removed on average 88% of the generated `AND` nodes and 67% of the edges.

In most cases we obtained exactly the TFPG we expected. This shows the adequacy the synthesis approach, in that TFPGs produced according to Problem 1 indeed correspond to what, intuitively, we would like to obtain. In some cases the synthesis procedure was able to derive further temporal constraints among the nodes given in input. In the Cassini use case, for instance, we discovered a strict precedence between two events that we didn't expect, but which we could confirm by inspection of the model.

From Figure 3.11a it can be seen that the number of given nodes (failure mode nodes plus discrepancies) clearly affects synthesis runtime, for some cases more, and for some less. However, the absolute number of input nodes is not the only factor for runtime. Figure 3.11b is a filter over Figure 3.11a, showing all synthesis result for a single system model (Cassini4). It can be seen that in several cases for the same number of input nodes very different runtimes are encountered, as for some sets of nodes the minimal cut-set computation is more difficult than for others.

Finally, we ran the graph correctness check on the synthesized and simplified TFPGs. The additional step of computing the precedence constraints of nodes in a given TFPG and comparing them to the precedence constraints in the system model is negligible in comparison to computing the precedence constraints in the system model; it took at most $2.3s$, and most cases where executed in less than $1s$.

3.8 Related Work

Traditionally, TFPGs received considerable interest as a basis for diagnosis implementations and system health management [Misra et al., 1992, Ofsthun and Abdelwahed, 2007, Abdelwahed et al., 2009]. In this thesis we are interested in techniques to assess how well TFPGs represent the faulty behaviors of a reference system model and in methods to compute TFPGs automatically from such models. We present the main results in this area.

[Strasser and Sheppard, 2011] develop a method to address the difficulty in building TFPGs with as few errors as possible, where errors are wrong relationships between faults, their effects, and the alarms. The technique for adding individual missing links or removing wrong links uses historical maintenance data coming from the actual implementation and computes probabilities for links between failure and monitoring events (graph nodes). The quality of the improvement depends on the quality of the data, and complete removal of errors cannot in general be guaranteed. Time bounds and mode constraints are also not considered by this technique. Finally, this method relies on a number of maintenance interventions on an implementation that record the alarms and identify through inspection the true cause of the alarms, and is thus not applicable at design time.

[Dubey et al., 2013] describe an approach to automatically generate TFPGs within a framework for component-based real-time software systems (ARINC-653 Component Model). Several types of failures and violations are defined for each component type, such as user-code failures and deadline violations, and the knowledge of how the whole assembly is structured is used to derive a corresponding TFPG. The synthesis approach relies on knowledge of local component behavior specified by data and control flow graphs to derive port and component-level TFPGs, and uses the structural knowledge to combine these local results in a TFPG covering the entire assembly. The structure of the TFPG thus reflects the functional component topology of the overall system. Relying on a specific component model, the method is applicable only to software systems implementing it.

A similar framework for TFPG synthesis is presented in [Priesterjahn et al., 2013]. It is also based on a component-based modeling framework (MechatronicUML), but here the components' behaviors are specified by timed automata as opposed to static information encoded in data and control flow graphs. The algorithm explores the behaviors of the timed automaton resulting from synchronization of all component automata, and aims to build TFPGs that relate failures in a component's input to discrepancies in its outputs. These relationships are discovered by comparing each path of the zone graph representing the nominal behavior to each path of all zone graphs of the behavior under predefined failure contexts. Also the time bounds on propagation delay are computed during the traversal. By encoding component behavior as a timed automaton it is possible to consider more complex behavioral patterns that emerge by comunication among components and sequences of state changes inside the components. However, this also makes TFPG synthesis more difficult compared to the structural approach in [Dubey et al., 2013]. No experimental evaluation of the approach is given and it is thus difficult to assess its scalability.

The approach to TFPG synthesis in [Priesterjahn et al., 2013] differs with our contribution in the following ways. First, we support generic finite and infinite-state transition systems as opposed to timed automata; this allows us for instance to deal with systems where time is modeled in a discrete fashion, or which have infinite-domain variables other than clocks. On the other hand we don't have continuous time in our framework, which is something we want to address in future work. We provide a number of formal TFPG properties that our validation and synthesis approaches refer to; in [Priesterjahn et al., 2013] instead no formal characterization of the synthesis result is given. Our synthesis framework is more generic in that no particular modeling approach and specification language is assumed. Failure modes and discrepancies are not bound to input and output ports of components, but can be general Boolean functions over the state variables of the system. Finally, by mapping the problem to the model-checking framework, state-of-the-art verification tools can be directly applied.

TFPGs are not the only formalism to study failure propagation. Besides FMEA and FTA, also *Bayesian networks* have been studied for this purpose (e.g. [Bobbio et al., 2001]), also for health management solutions that integrate TFPGs and Bayesian networks [Oonk and Maldonado, 2016]. Their focus however is on probabilistic modeling of failures and their effects, whereas TFPGs describe a system's temporal fault propagation in a multi-mode context.

Temporal-causal graphs are directed graphs that describe how various system parameters affect each other over time. They can be used to identify the propagation effects of faults based on the relationships between parameters, as done for instance in [Narasimhan and Biswas, 2007]. TFPGs instead model sequences of failure events and alarms with information on time delays between related pairs of events, and target thus a higher abstraction level.

A causality framework that points, like TFPGs, at a higher abstraction level is *causality checking* [Leitner-Fischer and Leue, 2013]. The definitions of causality given in [Halpern and Pearl, 2005] are adapted to finite-state transition systems, which enables the causal analysis of system evolutions over time. The goal of this analysis is to identify the minimal sets of events that lead to a specific top-level effect, including constraints on their ordering and on the absence of events that inhibit the effect. This framework can also be used to model and discover failure propagations as sequences of events. The event orders in these sequences however are specific for a chosen effect, i.e. for the traces leading to the top-level effect, and might not hold in general. TFPGs instead aim at representing an ordering among failure-related events that is valid for all possible executions, and additionally integrate this information with time delays and mode constraints between related events.

3.9 Summary

In this chapter we made several contributions that facilitate working with TFPGs as abstractions of a timed dynamic system model. TFPGs abstract the failure propagations in such a model in terms of discrete ordering of failure events and their Boolean combination, time bounds on the delays between these events, and mode constraints on the propagations. These features make them very useful as reference models for designing fault protection.

The developed techniques address two problems that are important for practical use of TFPGs. First, the issue of validating a given TFPG against a model of the system dynamics; validating the assumptions they represent is important if fault protection architectures are to be designed based on them. Second, the issue of generating a TFPG automatically from the system model; saving on modeling cost (and time) is crucial for improving their acceptance in real projects.

To adress these problems we developed a trace-based semantics for TFPGs to map system traces to corresponding TFPG traces. A definition of when a TFPG trace satisfies the TFPG constraint is given. Based on these results we introduced a technique based on model-checking to check whether all system traces comply with the constraints encoded by a given TFPG, and whether its edge parameters can be made more accurate. Finally we described a synthesis algorithm that first synthesizes the underlying graph of the TFPG, then removes redundant edges to make the TFPG more readable for engineers, and finally improves the parameters on edges.

The approach has been fully implemented using symbolic model-checking techniques for infinite-state transition systems. The advantage of this reduction to model-checking is that we can use – off-the-shelf – state-of-the-art tools as a reasoning back-end instead of ad-hoc implementations, and thus automatically benefit from advancements in the field. A thorough evaluation based on a number of syntetic and industrial benchmarks demonstrates the practicality of the approach. The results are encouraging, because in aerospace projects, for instance, the size of the TFPGs used in the tests corresponds to relevant problems at the unit to subsystem level. Positive feedback is reported also from the application in an industrial setting by the industrial partners in the FAME project [European Space Agency, 2011]. Further case studies on an application in an industrial context are described in Chapter 5.

There are several interesting directions for future work. First of all we would like to extend the framework and algorithms to dense-time models such as hybrid automata. This would allow to explicitly reason on the continuous evolution of real-valued variables and thus higher expressivity and a more natural modeling style for physical phenomena. The whole chain from definitions to algorithms would need to be lifted to the hybrid case, but we don't expect major problems for this line of work.

To improve scalability we will investigate TFPG analysis in a compositional framework based on assume-guarantee reasoning. Related results have recently been developed for Fault Tree Analysis in [Bozzano et al., 2014b], which studies ways to relate contract failures among various compontents in an architecture. It would however remain to be seen whether limiting the propagation analysis to contract violations is useful in practice, or if at least discrepancies need to refer to more concrete implementation details.

We would also like to investigate exploration of the solution space for TFPG synthesis. For instance, depending on the order in which parameters are tightened, different solutions might be obtained. The reason for this is that our interpretation of TFPG satisfaction is a local one focusing on individual nodes, and therefore situations can arise where different ways to obtain local consistency exist by differently balancing all parameter assignments in the graph. A challenge will be to identify metrics that can rank different solutions, and also what exactly the relevant dimensions to be investigated are. One interesting property to consider is diagnosability, and dimensions of interest could be simplification vs. tightening of edges, or also per-mode TFPG synthesis with subsequent merging of individual TFPGs.

Finally we note that our approach to validate and generate TFPGs is purely based on temporal reasoning, in the sense that we observe the temporal ordering of failure events without considering architectural structure of the underlying system (if available). This could result in propagation paths that reflect pure temporal correlations and are thus counter-intuitive to engineers. Investigating TFPG synthesis approaches that take into consideration also structural information might thus produce clearer TFPGs in such cases. In general, it could also be a way to improve synthesis performance through a bootstrapping step that creates a first version of the TFPG using only structural information. Care however needs to be taken not to discard also propagations between nodes that are not related from a functional point-of-view in terms of system functions, but which effectively have some relationship on a lower level.

Chapter 4

Diagnosability Analysis

A key element in an FDIR architecture are the monitors or alarms[1] used for the detection step. They signal the presence of faults and are used to trigger isolation and recovery procedures. What makes the activity of designing monitors challenging is the partial observability of the system's state. The faulty conditions and their effects cannot be observed directly, but have to be inferred from the available sensors and measurements. This process is often referred to as *diagnosis* [Reiter, 1987], and the component performing diagnosis and raising alarms or triggering monitors is usually called a *diagnoser*.

The task of verifying the feasibility of diagnosis based on the available observations is called diagnosability analysis, and is the focus of this chapter. After developing, in the previous chapter, a type of failure analysis that validates and automatically generates temporal models of failure event sequences, in this chapter we investigate diagnosability analysis as a failure analysis that studies whether a specific condition that needs to be diagnosed does indeed have a distinctive effect on the available observations.

We build on the framework of [Bozzano et al., 2014a] for the specification and generation of on-line model-based diagnosers that sample the system under diagnosis at constant intervals. The framework was developed within

[1]The terms "monitor" and "alarm" are used interchangeably.

the AUTOGEF and FAME projects of the European Space Agency (see [AUTOGEF, 2016] and [FAME, 2016]). It is very expressive, including temporally extended diagnosis conditions, various forms of delay, and is defined, based on temporal epistemic logic, on single points of individual traces. Delay constraints on the diagnosis can be used to require alarms being triggered before more severe conditions occur.

Within this framework, we consider two fundamental problems in diagnosability. The first problem, *verification of diagnosability*, amounts to checking whether the available sensors are sufficient to infer the desired information on the hidden behavior of the system, for instance the presence of faults. Verification of diagnosability (also known as "diagnosability verification", or "diagnosability checking") may confirm that the choice of sensors is adequate or may pinpoint deficiencies of the choice. The second problem, *synthesis for diagnosability*, aims at identifying subsets of the available sensors that are sufficient to ensure diagnosability, while possibly minimizing some cost function. Synthesis for diagnosability (also known as "synthesis of observability requirements", "sensor placement", and "sensor selection for diagnosability") may support the design process by automatically devising the most suitable choice of sensors, among many possible choices, thus reducing overall costs.

We remark the difference between diagnosability and the problem of *diagnoser effectiveness* as described in [Bozzano et al., 2015c]. Diagnosability is a property of a partially observable plant, i.e. whether the information conveyed by the available sensors is sufficient (for an *ideal* diagnoser) to carry out a given diagnosis task. The analysis of diagnosability is a fundamental phase during the design of the plant: if the plant is not diagnosable, then it is impossible to build a diagnoser for it. Diagnoser effectiveness, instead, is the problem of verifying whether a *given* diagnoser behaves as expected. This problem can be directly seen as a problem of model-checking the properties of the composition of the plant and the diagnoser. In some sense, diagnosability is a more difficult problem, because it says something on *all* possible diagnosers. Diagnoser effectiveness is out of the scope of this chapter.

We develop the following contributions.

1. First we extend the framework with the notion of context. "Universal" diagnosability in all possible operational contexts and along all possible system evolutions is very difficult to achieve, and with contexts we provide a means to specify what behaviors must be diagnosable. We encode the conditions under which we can expect the FDIR to operate and be able to infer the required information. For example, it may be unreasonable to expect that a faulty condition is detected if an arbitrary large number of concurrent faults can happen; another common assumption is to limit the number of faults occurring at the same time (e.g. single-fault assumption).

2. The second key contribution is a practical algorithm for diagnosability verification and an analysis of its theoretical properties. The classical way to falsify diagnosability is to identify specific pairs of traces called *critical pairs* [Jiang et al., 2001, Cimatti et al., 2003]. These consist of two executions that are observationally indistinguishable but only one of them contains a condition that should be detected. The existence of a critical pair results in the impossibility for a diagnoser to ascertain whether the condition actually occurred.

   With the increased expressiveness of the framework however the task of proving diagnosability becomes more complex. We show how the absence of critical pairs is both a necessary and sufficient condition for diagnosability in many important cases, and describe what information we obtain from the check in cases where it is only a necessary condition.

We show how the existence of a critical pair for a given plant can be reduced to checking whether a suitable temporal formula, expressed in LTL, holds over the corresponding twin-plant. The twin-plant consists of two replicas of the original system model, so as to encode the space of indistinguishable traces and to produce critical pairs. This reduction approach has distinctive advantages over using epistemic model-checking as in [Bozzano et al., 2014a]; for instance it allows us to leverage recent developments in temporal logic model-checking for finite- and infinite-state models. Further advantages are discussed in more detail in Section 4.6.

3. The third contribution is an effective synthesis algorithm for diagnosability, able to produce all sensor configurations that ensure diagnosability, and are possibly minimal (with respect to set inclusion) or optimal (with respect to a given cost function). We achieve this by building a *parameterized twin-plant*, that can be seen as a symbolic representation of the mapping between the space of sensor configurations and the corresponding plants. The idea is to use a single parameterized model, where each parameter models the availability of a specific sensor. The algorithm exploits the symbolic representation of the space of sensor configurations to prune the search based on monotonicity considerations. Intuitively, if a sensor configuration is not sufficient for diagnosability, none of its subsets can be.

   Our synthesis approach differs from the state-of-the-art approaches like [Grastien, 2009] that use an enumerative strategy guided by the cost function. In early design phases, cost functions might not be available, thus the ability to reason on the whole sensor configuration space and perform "what-if" analysis provides an additional value. Furthermore, our benchmarks show that even when having information on the costs, the enumerative approach is not guaranteed to perform better. Finally, by directly leveraging a parameter-synthesis engine, we can benefit from the continuous improvements from the model-checking community without the need of developing domain-specific algorithms.

All techniques for diagnosability were implemented within the xSAP platform, with nuXmv as a back-end. We carried out an evaluation on a comprehensive set of realistic benchmarks from various application domains, in part shared with the experiments in Section 3.7. The results demonstrate the effectiveness of the approach on benchmarks of industrial size. Partial results were published in [Bittner et al., 2012] (parameterized twin-plant) and [Bittner et al., 2014a] (cost-driven parameter synthesis applied to diagnosability), and a journal article summarizing the chapter is being prepared for submission.

The chapter is structured as follows. In Section 4.1 we define the problem of diagnosability verification. In Section 4.2 we present the twin-plant construction for diagnosability checking. In Section 4.3 we describe the problem of synthesis for diagnosability. In Section 4.4 we describe the symbolic parameterized twin-plant representation, and discuss the algorithms for diagnosability synthesis. In Section 4.5 we experimentally evaluate the proposed methods on a set of benchmarks from various domains. In Section 4.6 we discuss the related work. In Section 4.7 we draw some conclusions and outline directions for future activities.

4.1 Verification: Problem Definition

In this section we define the problem of verification of diagnosability. We first define the framework for diagnoser specification, and then discuss the

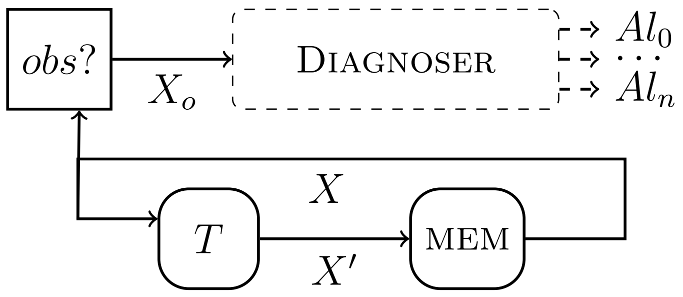


Figure 4.1: A plant and its diagnoser.

notion of diagnosability.

4.1.1 Specifying a Diagnoser

Diagnosability is best understood in the context of the design of a diagnoser. Consider Figure 4.1. The lower part, with solid lines, represent the plant: T is the transition relation that maps the (assignment to the) current state variables X to the (assignment to the) next state variables X'; MEM is the memory element, transforming the next state to the current one in response to the clock tick. The *obs*? block defines the observable variables. The upper part of Figure 4.1, with dashed lines, represents the FDI component, also referred to as diagnoser. The diagnoser is a module that runs in parallel to the plant, driven by the observable variables X_o, and raises some alarms $Al_0, \ldots, Al_n$ in correspondence to events of interest, called diagnosis conditions. The behaviour of the diagnoser is typically specified by a set of requirements, describing the relationship between the alarms to be raised and the occurrence of the corresponding diagnosis conditions. It is in principle possible that the requirements are impossible to satisfy, for example because the available sensors do not convey sufficient information to appropriately raise the alarms in a correct and timely fashion. Verification of diagnosability can be intuitively seen as a form of requirements validation, i.e. checking whether the requirements of the diagnoser can indeed be realized.

Assumptions

The following assumptions are used in the rest of the chapter. Given is always a system under diagnosis modeled as a transition system, whose diagnosability we want to analyze. A context specification can be used to limit the analysis to behaviors of interest. As we build upon the results of [Bozzano et al., 2014a], we adopt the LTL view of time, i.e. every transition of the system corresponds to one unit of time or one tick passing. The diagnoser is a deterministic transition system having access to the observable signals of the plant. The diagnoser and the plant are synchronously connected, i.e. for each transition of the plant there is a corresponding transition of the diagnoser. The diagnoser does not influence the behaviour of the system (we consider active diagnosis [Sampath et al., 1998] beyond the scope of this chapter). The diagnoser has perfect recall, i.e. it is able to keep track of all the history of the plant. This is the general assumption in the literature on diagnosability, and it is opposed to bounded recall (see e.g. [Gario, 2016]).

Diagnosis Conditions

The central element for the specification of an FDI requirement is the *diagnosis condition* to be monitored, denoted with β. In order for the diagnosis condition to be evaluated by a diagnoser, we require that β encodes a property on the past, i.e., given a prefix of a trace π^k, we can say whether β is satisfied in $\pi[k]$ or not. Formally, a *diagnosis condition* for S is any formula β built according to the following rule: $\beta ::= p \mid \beta_1 \wedge \beta_2 \mid \neg\beta \mid \mathsf{Y}\beta \mid \beta_1 \mathsf{S} \beta_2$ where p is an atomic proposition over the state variables X.

In this formalism, we can express complex diagnosis conditions (see Figure 4.2). First, it is possible to model fault detection (whether any fault

$$\begin{aligned}
\beta_1 &\doteq (f_1 \vee \ldots \vee f_n) \\
\beta_2 &\doteq (f_3) \\
\beta_3 &\doteq (f_2 \wedge \mathsf{O}^{\leq 3} f_1) \\
\beta_4 &\doteq \mathsf{H}^{\leq 10}(Heat \ \wedge \ |t - t_{prev}| \leq \epsilon)
\end{aligned}$$

Figure 4.2: Examples of Diagnosis Conditions

is present), with $\beta \doteq (f_1 \vee \ldots \vee f_n)$, and fault identification (which of the possible faults is present), with $\beta \doteq (f_i)$ for a specific i. We might also want to restrict the detection to a particular sub-system, or identification among two similar faults might not be of interest. Second, it is possible to express sequences of relevant situations, e.g. f_2 preceded of up to three ticks by f_1, by stating $\beta \doteq (f_2 \wedge \mathsf{O}^{\leq 3} f_1)$. Finally, it is possible to define as diagnosis conditions for an infinite-state system some relation between the values of a real-valued variable over time. For example, with $\beta \doteq \mathsf{H}^{\leq 10}(Heat \ \wedge \ |t - t_{prev}| \leq \epsilon)$ we define a diagnosis condition where t does not change more than a small amount ϵ for 10 cycles even if the heater is on, with t_{prev} being a variable recording the value of t in the previous state.

Alarm Conditions

Besides the diagnosis condition, representing the property we want to diagnose, we are interested in the temporal *delay* between the occurrence of this condition and the raising of an associated alarm. Indeed it is realistic to assume that faults can go undetected for a certain amount of time, and clearly stating how long this interval can be at most is crucial for guaranteeing real-time properties of the overall fault protection solution. The specific bound on diagnosis delay can for instance be derived from a preceding TFPG analysis (see Chapter 3); when a propagation between two nodes needs to be avoided, a monitor should trigger within at most t_{min} time units to guarantee detection before the unwanted propagation (recovery time not included).

In this chapter, we follow [Bozzano et al., 2015c] and consider three kinds of *alarm conditions*, which we denote with $\textsc{ExactDel}(Al, \beta, d)$, $\textsc{BoundDel}(Al, \beta, d)$, and $\textsc{FiniteDel}(Al, \beta)$:

1. $\textsc{ExactDel}(Al, \beta, d)$ specifies that whenever β is true, Al must be triggered exactly d steps later and Al can be triggered only if d steps earlier β was true. Formally, for any trace π of the system, if β is true along π at the time point i, then Al is true in $\pi[i + d]$ (completeness); if Al is true in $\pi[i]$, then β must be true in $\pi[i - d]$ (correctness).

2. $\textsc{BoundDel}(Al, \beta, d)$ specifies that whenever β is true, Al must be triggered within the next d steps and Al can be triggered only if β was true within the previous d steps. Formally, for any trace π of the system, if β is true along π at the time point i then Al is true in $\pi[j]$, for some $i \leq j \leq i + d$ (completeness); if Al is true in $\pi[i]$, then β must be true in $\pi[j']$ for some $i - d \leq j' \leq i$ (correctness).

3. $\textsc{FiniteDel}(Al, \beta)$ specifies that whenever β is true, Al must be triggered in a later step and Al can be triggered only if β was true in some previous step. Formally, for any trace π of the system, if β is true along π at the time point i then Al is true in $\pi[j]$ for some $j \geq i$ (completeness); if Al is true in $\pi[i]$, then β must be true in $\pi[j']$ for some $0 \leq j' \leq i$ (correctness).

Figure 4.3 shows an example of admissible responses for the various alarms to the occurrences of the same diagnosis condition β. In the case of $\textsc{BoundDel}(Al, \beta, 4)$ the alarm can be triggered at any point as long as it is within the next 4 time-steps. This is quite different from having 5 different

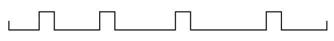
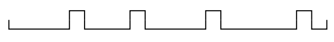
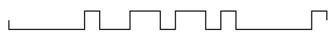
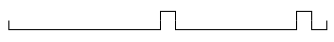
β	
$\textsc{ExactDel}(Al, \beta, 2)$	
$\textsc{BoundDel}(Al, \beta, 4)$	
$\textsc{FiniteDel}(Al, \beta)$	

Figure 4.3: Examples of alarm responses to the diagnosis condition β.

	LTL Formulation	
Alarm Condition	Correctness	Completeness
$\textsc{ExactDel}(Al, \beta, d)$	$\mathsf{G}(Al \rightarrow \mathsf{Y}^d \beta)$	$\mathsf{G}(\beta \rightarrow \mathsf{X}^d Al)$
$\textsc{BoundDel}(Al, \beta, d)$	$\mathsf{G}(Al \rightarrow \mathsf{O}^{\leq d} \beta)$	$\mathsf{G}(\beta \rightarrow \mathsf{F}^{\leq d} Al)$
$\textsc{FiniteDel}(Al, \beta)$	$\mathsf{G}(Al \rightarrow \mathsf{O} \beta)$	$\mathsf{G}(\beta \rightarrow \mathsf{F} Al)$

Figure 4.4: ASL Alarm conditions as LTL formulae

alarms $\textsc{ExactDel}(Al_0, \beta, 0)$, ..., $\textsc{ExactDel}(Al_4, \beta, 4)$ and considering their disjunction $(Al_0 \vee \cdots \vee Al_4)$, since the latter requires that we are always able to exactly pin-point the moment in which the diagnosis condition occurred, while the former provides a degree of flexibility. Similarly, $\textsc{FiniteDel}(Al, \beta)$ is of particular theoretical interest since it captures the idea that, in some systems, the delay might be finite but unbounded.

The meaning of the alarm conditions is further clarified in Figure 4.4, where we associate to each alarm condition type an LTL formalization encoding the concepts of correctness and completeness. The first conjunct expresses *correctness*, and intuitively says that whenever the diagnoser raises an alarm, then the fault must have occurred. *Completeness*, the second conjunct, intuitively encodes that whenever the fault occurs, the alarm will be raised.

We denote *alarm conditions* with Ξ. An alarm condition constrains the behavior of the diagnoser to respond to events that occur in the plant. These are typically non-observable events, and thus the diagnoser may need to infer their occurrence by only relying on the observable behavior of the plant. Thus, when we say that a diagnoser D for S satisfies an alarm condition Ξ, we mean that the composition between D and S satisfies the LTL formula corresponding to Ξ in Figure 4.4.

A *context* constrains the operational setting of the environment in which we place the plant. Consider the situation in which multiple concurrent faults are extremely unlikely to happen. From the engineering point of view, it might make sense to study the diagnosability of the system under a single-fault assumption. For this reason we introduce the concept of context (denoted by Ψ) as a subset of the possible traces of the plant (i.e. $\Psi \subseteq \Pi(S)$). Note that it makes little sense to propagate this operational choice within the system model and thus to blur the line between system modeling and operational requirements. In many cases indeed we are not dealing with a restriction of the model under analysis, but an assumption on an external influence such as a controller or the environment, which are by definition not part of the system being modeled.

We remark that the framework is very rich: it encompasses infinite-state systems, temporally extended monitoring conditions over states and events, various forms of delay, and an operational context. In fact, it captures the main forms of diagnosability in the literature. The expressiveness of the framework is discussed in greater detail, from a technical standpoint, in Section 4.6.

4.1.2 From Diagnosability to Critical Pairs

Let an alarm condition, also referred to as a *diagnoser specification*, be given. The problem of verification of diagnosability amounts to checking whether it is possible for any diagnoser to satisfy the diagnoser specification. Following [Bozzano et al., 2015c], which adopts an epistemic point-of-view on diagnosis and diagnosability, we provide a definition of diagnosability that is more fine-grained compared to related work, that is, as a property of specific points of system traces. We also show how this can be generalized to whole traces and to sets of traces. Furthermore, compared to [Bozzano et al., 2015c], we use a richer notion of diagnosability relative to an operational context Ψ. In the following we assume that alarm conditions are analyzed individually, and replace the alarm signal Al with $\circ$.

Definition 14. *Let S be a plant, $\Xi := \textsc{ExactDel}(\circ, \beta, d)$ an alarm condition, $\Psi \subseteq \Pi(S)$ a context, a trace $\pi_1 \in \Psi$ and i a trace index. We say that Ξ is* trace diagnosable *in $\langle \pi_1, i\rangle$ w.r.t. Ψ iff $\pi_1, i \models \beta \Rightarrow (\forall \pi_2 \in \Psi \cdot obs(\pi_1^{i+d}) = obs(\pi_2^{i+d}) \Rightarrow \pi_2, i \models \beta)$.*

Definition 15. *Let S be a plant, $\Xi := \textsc{BoundDel}(\circ, \beta, d)$ an alarm condition, $\Psi \subseteq \Pi(S)$ a context, a trace $\pi_1 \in \Psi$ and a trace index i. We say that Ξ is* trace diagnosable *in $\langle \pi_1, i\rangle$ w.r.t. Ψ iff $\pi_1, i \models \beta \Rightarrow (\exists j \in \mathbb{N}\ i \leq j \leq i+d \cdot \forall \pi_2 \in \Psi \cdot obs(\pi_1^{j}) = obs(\pi_2^{j}) \Rightarrow \exists k \in \mathbb{N}\ j-d \leq k \leq j \cdot \pi_2, k \models \beta)$.*

Definition 16. *Let S be a plant, $\Xi := \textsc{FiniteDel}(\circ, \beta)$ an alarm condition, $\Psi \subseteq \Pi(S)$ a context, a trace $\pi_1 \in \Psi$ and i a trace index. We say that Ξ is* trace diagnosable *in $\langle \pi_1, i\rangle$ w.r.t. Ψ iff $\pi_1, i \models \beta \Rightarrow (\exists j \in \mathbb{N}\ i \leq j \cdot \forall \pi_2 \in \Psi \cdot obs(\pi_1^{j}) = obs(\pi_2^{j}) \Rightarrow \exists k \in \mathbb{N}\ k \leq j \cdot \pi_2, k \models \beta)$.*

With these definitions we can say something about a specific point of a specific trace. We now extend them to sets of traces, always relative to a context Ψ.

- An alarm condition is trace diagnosable on a trace $\pi \in \Psi$ w.r.t. Ψ iff it is trace diagnosable w.r.t. Ψ in $\langle \pi, i\rangle$ for all $i \in \mathbb{N}$.
- An alarm condition is system diagnosable w.r.t. Ψ iff for all $\pi \in \Psi$ it is trace diagnosable on π w.r.t. Ψ.

- An alarm condition is system diagnosable (i.e. diagnosable in the classical sense of [Sampath et al., 1996]) iff it is system diagnosable in $\Psi = \Pi(S)$ (i.e. on all traces of the plant).

We now relate the diagnosability property to the existence of a pair of traces of the plant, referred to as *critical pair*. These pairs of traces are the standard way to falsify diagnosability since [Jiang et al., 2001, Cimatti et al., 2003], as looking for them is an easier way to prove or disprove diagnosability than building a diagnoser (e.g. [Sampath et al., 1995]). We adapt the concept to each type of alarm pattern.

Definition 17 (Critical Pair)**.** *Let S be a plant, β a diagnosis condition, and $\Psi \subseteq \Pi(S)$ a context. We say that $\pi_1, \pi_2 \in \Psi$ are a critical pair for an alarm condition Ξ at time i iff $\pi_1, i \models \beta$ and:*

- *if $\Xi \doteq \textsc{ExactDel}(\circ, \beta, d)$, then $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$ and $\pi_2, i \not\models \beta$;*
- *if $\Xi \doteq \textsc{BoundDel}(\circ, \beta, d)$, then $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$, and $\pi_2, j \not\models \beta$ for all $j \in [i-d, i+d]$;*
- *if $\Xi \doteq \textsc{FiniteDel}(\circ, \beta)$, then $obs(\pi_1) = obs(\pi_2)$, and $\pi_2, j \not\models \beta$ for all j.*

We say that $\pi_1, \pi_2 \in \Psi$ are a critical pair for Ξ if it is one for some time point $i \in \mathbb{N}$.

In [Sampath et al., 1995] the notion of critical pair is used to define the idea of diagnosability. In our framework, we use a richer notion of diagnosability, and therefore, we need to discuss the relation between the existence of a critical pair and the diagnosability of the system. In the next two sections we will show that while the existence of a critical pair is a sufficient condition for non-diagnosability, in some cases it is not a necessary condition.

Diagnosability Verification via Critical Pairs: Correctness

First we establish the correctness of the critical pair approach of verifying diagnosability. Correctness requires that when a critical pair is found, the respective alarm condition must indeed not be diagnosable for the given system. This is straight-forward in all cases.

Theorem 8 (Critical Pairs: Correctness). *Let S be a plant, Ξ an alarm condition, $\Psi \subseteq \Pi(S)$ a context, $\pi_1 \in \Psi$ a trace, and $i \in \mathbb{N}$ a trace index. If a trace $\pi_2 \in \Psi$ exists such that π_1, π_2 are a critical pair for Ξ at time i as per Definition 17, then Ξ is not trace diagnosable in π_1, i w.r.t. Ψ.*

Proof. Assume there exists a trace $\pi_2 \in \Psi$ such that π_1, π_2 are a critical pair for Ξ at time i, but Ξ is trace diagnosable in π_1, i w.r.t. Ψ. We show for each type of pattern that this leads to a contradiction.

$\Xi :=$ **ExactDel(**$\circ, \beta, d$**)** From Definition 17 we know that $\pi_1, i \models \beta$, $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$, and $\pi_2, i \not\models \beta$. Such a pair of traces is however not possible if Ξ is diagnosable, since, according to Definition 14, $\pi_1, i \models \beta$ and $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$ imply that $\pi_2, i \models \beta$, which is not true on the critical pair.

$\Xi :=$ **BoundDel(**$\circ, \beta, d$**)** From Definition 17 we know that $\pi_1, i \models \beta$, $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$, and $\forall j \in [i-d, i+d] \cdot \pi_2, j \not\models \beta$. Such a pair of traces is however not possible if Ξ is diagnosable, since, according to Definition 15, $\pi_1, i \models \beta$ implies that $\exists j \in [i, i+d]$ such that either $obs(\pi_1^j) \neq obs(\pi_2^j)$, which is not true for any j by construction of the critical pair, or $\exists k \in [j-d, j] \cdot \pi_2, k \models \beta$, which is also not true, as k is by definition within $[i-d, i+d]$, and we know that for none of these time points β holds on π_2.

$\Xi :=$ **FiniteDel(**$\circ, \beta$**)** From Definition 17 we know that $\pi_1, i \models \beta$, $obs(\pi_1) = obs(\pi_2)$, and $\forall j \in \mathbb{N} \cdot \pi_2, j \not\models \beta$. Such a pair of traces is however not possible if Ξ is diagnosable, since, according to Definition 16, $\pi_1, i \models \beta$ implies that $\exists j \in \mathbb{N} \cdot i \leq j$ such that either $obs(\pi_1^j) \neq obs(\pi_2^j)$, which is not true for any j by construction of the critical pair, or $\exists k \in [0, j] \cdot \pi_2, k \models \beta$, which is also not true for whatever j we choose.

□

Diagnosability Verification via Critical Pairs: Completeness

We now investigate the completeness of the critical pair method. By completeness we mean that whenever an alarm condition is diagnosable, we would like to prove it by proving the absence of critical pairs. For ExactDel this property is shown as follows.

Theorem 9 (Critical Pairs: Completeness for ExactDel)**.** *Let S be a plant, $\Xi :=$ ExactDel$(\circ, \beta, d)$ an alarm condition, and $\Psi \subseteq \Pi(S)$ a context. If Ξ is not system diagnosable w.r.t. Ψ, then a critical pair as per Definition 17 must exist in Ψ.*

Proof. Assume Ξ is not system diagnosable w.r.t. Ψ. Following Definition 14 and the definition of being system diagnosability w.r.t. a context, this means that for some $\pi_1 \in \Psi$ and some trace index $i \in \mathbb{N}$ we have $\pi_1, i \models \beta$ and $\exists \pi_2 \in \Psi$ s.t. $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$ and $\pi_2, i \not\models \beta$. These π_1, π_2 match the critical pair definition for ExactDel in Definition 17.

□

For BoundDel and FiniteDel, the relationship between diagnosability and the existence of critical pairs is more complex. We discuss both cases separately.

BoundDel We show now an example where the absence of critical pairs for a given d does not prove diagnosability with the same d. This problem exists only with delays $d > 0$, because with $d = 0$ we have that

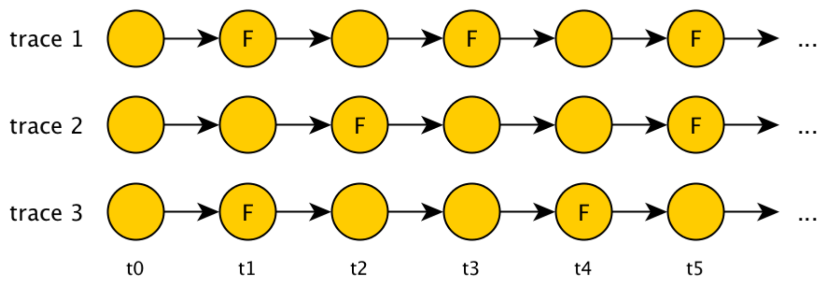


Figure 4.5: $\textsc{BoundDel}(\circ, F, 1)$ is not system diagnosable in this system, but no critical pair exists for that delay.

$\textsc{BoundDel}(\circ, \beta, 0)$ and $\textsc{ExactDel}(\circ, \beta, 0)$ express identical constraints on the traces of Ψ (see e.g. Figure 4.4).

Assume the traces shown in Figure 4.5 are the only traces in some plant S, that our context of interest includes all of them, and that they are all observationally indistinguishable. The diagnosis condition of interest is the proposition F, which marks faulty states; we assume that only the shown states are marked with F. The alarm condition $\textsc{BoundDel}(\circ, F, 2)$ is trace diagnosable in this example in all states of traces marked with F. Furthermore, $\textsc{BoundDel}(\circ, F, 1)$ is not system diagnosable, with a violation occurring in trace 1 at time $t3$. However, from these traces it is not possible to construct a $\textsc{BoundDel}$ critical pair for delay $d = 1$, only for $d = 0$.

It is important to note here that the problem is not in the specific definition of $\textsc{BoundDel}$ critical pairs, but the fact that falsification is done only considering *pairs* of traces. By the increased expressiveness of the framework to define alarm conditions, in the case of Figure 4.5 a witness for non-diagnosability of $\textsc{BoundDel}(\circ, F, 1)$ in trace 1 at point $t3$ consists of all three traces, whereas with only two traces diagnosability cannot be disproved.

Intuitively, the reason for this mismatch between the delay of the alarm condition and the delay of corresponding critical pairs is that with $\textsc{BoundDel}$

critical pairs we need to look backwards and forwards in order not to miss occurrences of β. A critical pair thus requires the absence of β in a diameter of $2d$ in the second trace, relative to the point i in the first one. The absence of BoundDel critical pairs for a delay d does however prove diagnosability with a delay bound $d' = 2d$. Without making further assumptions a lower bound cannot be guaranteed from the absence of critical pairs, even a $d' = 2d - 1$. An example of this is again the system of Figure 4.5.

Theorem 10 (Critical Pairs: Completeness for BoundDel (Upper Bound)). *Let S be a plant, $\Xi := \text{BoundDel}(\circ, \beta, 2d)$ an alarm condition, and $\Psi \subseteq \Pi(S)$ a context. If no critical pair $\pi_1, \pi_2 \in \Psi$ for $\Xi' := \text{BoundDel}(\circ, \beta, d)$ exists (Definition 17), then Ξ is system diagnosable w.r.t. Ψ.*

Proof. Assume no critical pair for Ξ' exists, but Ξ is not diagnosable. The fact that Ξ is not diagnosable implies that $\exists \pi_1 \in \Psi$, $\exists i \in \mathbb{N}$ such that $\pi_1, i \models \beta$ and, furthermore, $\forall j \in [i, i + 2d] \cdot \exists \pi_2 \in \Psi$ such that $obs(\pi_1^j) = obs(\pi_2^j)$ and $\forall k \in [j - 2d, j] \cdot \pi_2, k \not\models \beta$. In particular, for π_1 and i we pick $j = i + d$, for which we know exists a $\pi_2 \in \Psi$ such that $\pi_1, i \models \beta$, $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$, and $\forall k \in [i - d, i + d] \cdot \pi_2, k \not\models \beta$. π_1, π_2 are thus a critical pair for Ξ' in Ψ, which contradicts the initial assumption that no such pair exists in Ψ.

□

We now introduce an assumption under which the absence of critical pairs for some $\Xi := \text{BoundDel}(\circ, \beta, d)$ indeed proves the diagnosability of that Ξ. Specifically, we restrict ourselves to cases where β is monotonic, that is, for some reason it holds that on any point i of any trace $\pi \in \Psi$, if β holds at time i, then it also holds at any time $j \geq i$. The monotonicity assumption is quite common in the existing literature. In particular, [Sampath et al., 1995] (and all successive works based on it) the diagnosis condition is defined as the occurrence of a fault event at some point in the past, i.e. $\beta := \mathsf{O}f$ for some proposition f, which has monotonic behavior. As proven in [Bozzano et al., 2015c], this can be reduced to a BoundDel problem. From a practical point of view, all permanent faults have a monotonic behavior. Moreover, monotonicity is not limited to the past occurrence of fault events. Indeed, all diagnosis conditions of the form $\beta := \mathsf{O}\beta'$, such as the ones described in [Jéron et al., 2006], are monotonic.

Theorem 11 (Critical Pairs: Completeness for BoundDel (Monotonicity))**.** *Let S be a plant, $\Xi :=$ BoundDel$(\circ, \beta, d)$ an alarm condition, and $\Psi \subseteq \Pi(S)$ a context. Furthermore we assume that β is monotonic in S, i.e. for every trace $\pi \in \Psi$ we have $\pi \models \mathsf{G}(\beta \rightarrow \mathsf{G}\beta)$. If Ξ is not system diagnosable w.r.t. Ψ, then a critical pair as per Definition 17 must exist in Ψ.*

Proof. If Ξ is not system diagnosable w.r.t. Ψ, then for some $\pi_1 \in \Psi$ and some $i \in \mathbb{N}$ we have that $\pi_1, i \models \beta$, and $\forall j \in [i, i+d] \exists \pi_2 \in \Psi$ such that $obs(\pi_1^j) = obs(\pi_2^j)$ and $\forall k \in [j-d, j] \cdot \pi_2, k \not\models \beta$. From this we know that there exists a trace $\pi_2 \in \Psi$ (specifically, for $j = i + d$) such that we have $obs(\pi_1^{i+d}) = obs(\pi_2^{i+d})$ and $\forall k \in [i, i+d] \cdot \pi_2, k \not\models \beta$. From the monotonicity of β it furthermore follows that $\forall k \in [i-d, i+d] \cdot \pi_2, k \not\models \beta$. From these properties it follows that π_1, π_2 is a critical pair for Ξ.

□

FiniteDel Also in the case of FiniteDel the absence of critical pairs does not necessarily prove diagnosability. Imagine a situation where the context encodes a fairness constraint that causes the observations after a failure to always diverge *eventually*, but that this event can be delayed indefinitely. In this case it is impossible to find a pair of traces that are forever observationally equivalent, but where one has the fault and the other one does not. A witness for non-diagnosability here indeed would consist of an infinite set of traces.

For infinite-state systems, even if Ψ denotes $\Pi(S)$, we can construct an example in which for any given d we can find a pair of traces that require a delay $d+1$ to be diagnosable. Consider for instance a system in which a variable c decreases at each step (e.g., $c' = c/2$) after a failure occurs and that the fault's detection requires c to reach a certain fixed threshold. By choosing an initial value for c that is arbitrarily high, we can postpone for arbitrarily many steps the distinguishability of both traces.

However, if we require S to be finite state and the Ψ to be characterized by an invariant property, then the absence of critical pairs does indeed prove FiniteDel diagnosability.

Theorem 12 (Critical Pairs: Completeness for FiniteDel)**.** *Let S be a finite state plant, $\Xi :=$ FiniteDel$(\circ, \beta)$ an alarm condition. Let $\Psi \subseteq \Pi(S)$ be the set of traces that satisfies a corresponding invariant property. If Ξ is not system diagnosable w.r.t. Ψ, then there must exist a critical pair for Ξ according to Definition 17.*

Proof. We prove the theorem by constructing a critical pair for Ξ. We assume that $\beta := p$ for some proposition p. All other cases can be reduced to this case by extending S with a monitor tracking β to identify whether it holds in any given point of a trace. The same strategy of introducing monitors is used in [Bozzano et al., 2014a].

- From the non-diagnosability of Ξ we know that there exists a $\pi_1 \in \Psi$ and $\exists i \in \mathbb{N}$ such that Ξ is not trace diagnosable in π_1, i.
- We then pick a j such that $j \geq i + N \times N$, that is, we consider at least $N \times N$ more steps in π_1 after the point i, where N is the size of the state space of S.
- According to the definition of diagnosability this means that there must be another trace $\pi_2 \in \Pi(S)$ such that $obs(\pi_1^j) = obs(\pi_2^j)$ and for

all $k \in \mathbb{N}$ with $k \leq j$ we have $\pi_2, k \not\models \beta$.

- As the state space of S is finite, of dimension N, there can be at most $N \times N$ combinations of pairs of states, which means that at some point $l \in \mathbb{N}$ with $i < l \leq j$ the pair $(\pi_1[l], \pi_2[l])$ has already been seen before, i.e. there is some $l' \in \mathbb{N}$ with $i \leq l' < l$ such that $\pi_1[l'] = \pi_1[l]$ and $\pi_2[l'] = \pi_2[l]$.
- From π_1 we now extract a new trace $\pi_{1'} := s_0, i_1, s_1, \ldots, i_{l'}, s_{l'}, (i_{l'+1}, s_{l'+1}, \ldots, i_l, s_l)^\omega$. The part between parentheses is the loop that is repeated ad infinitum.
- Note that $\pi_{1'}$ is also a trace belonging to Ψ, because the invariant property defining Ψ is true in every state of it by virtue of being built from π_1.
- The same procedure is performed for π_2, obtaining another trace $\pi_{2'} \in \Psi$.
- Both $\pi_{1'}$ and $\pi_{2'}$ have the same observations up to $l-1$ and then loop back to the same synchronization point l', hence $obs(\pi_{1'}) = obs(\pi_{2'})$.
- Furthermore by construction from π_1 we have $\pi_{1'}, i \models \beta$.
- Finally, it holds that for all $k \in \mathbb{N}$ we have $\pi_2' \not\models \beta$.
- $\pi_{1'}, \pi_{2'}$ thus are a critical pair for Ξ.

□

This result can be generalized to any context representing a safety property, since it can be encoded as an invariant by extending the original plant with a corresponding monitor – see for instance [Baier and Katoen, 2008] for finite-state systems, which we are interested in in Theorem 12.

The reasoning done in the proof of Theorem 12 does not apply if the traces in Ψ are characterized by a liveness property, because such properties can be falsified only on infinite traces. Detecting violation or satisfaction of the property on a finite prefix, e.g. with a monitor, is thus not possible, and we cannot construct the traces $\pi_{1'}$ and $\pi_{2'}$ in a way that guarantees they are still traces of Ψ. The proof strategy also does not work for infinite-state systems, where identifying loops as done here might not even be possible.

We show now several monotonicity properties of alarm conditions that can be used to say something about the diagnosability of related alarm conditions.

Theorem 13 (Monotonicity)**.** *The following conditions hold for all S, β, Ψ.*

1. *if some Ξ is diagnosable relative to context Ψ, then so it is relative to all contexts $\Psi' \subset \Psi$*
2. *if* $\textsc{ExactDel}(\circ, \beta, d)$ *is diagnosable,*
   then so is $\textsc{BoundDel}(\circ, \beta, d)$
3. *if* $\textsc{BoundDel}(\circ, \beta, d)$ *is diagnosable for some d,*
   then so is $\textsc{FiniteDel}(\circ, \beta)$
4. *if* $\textsc{ExactDel}(\circ, \beta, d)$ *is diagnosable for some d,*
   then so is $\textsc{ExactDel}(\circ, \beta, d')$ *for all $d' > d$*
5. *if* $\textsc{BoundDel}(\circ, \beta, d)$ *is diagnosable for some d,*
   then so is $\textsc{BoundDel}(\circ, \beta, d')$ *for all $d' > d$*

Proof. We rely on Definitions 14, 15, and 16 of trace diagnosability to show these properties by case.

1. Assume Ξ is diagnosable in Ψ but not in Ψ'. Since $\Psi' \subset \Psi$, if a set of traces disproving it exists in Ψ', then it must also exist in Ψ, and Ξ is not diagnosable in Ψ, reaching a contradiction.

2. Pick any $\pi_1 \in \Psi$ and any $i \in \mathbb{N}$ such that $\pi_1, i \models \beta$. Since $\textsc{ExactDel}(\circ, \beta, d)$ is trace diagnosable in π_1, i, the j required by $\textsc{BoundDel}$ can be set to $j = i + d$, because for all other traces $\pi_2 \in \Psi$ such that $obs(\pi_1^j) = obs(\pi_2^j)$ we can set the k required by $\textsc{BoundDel}$ to $k = i$, since from $\textsc{ExactDel}$ diagnosability we know that $\pi_2, k \models \beta$.

3. Similarly, for any $\textsc{BoundDel}$ trace diagnosable π_1, i such that $\pi_1, i \models \beta$, the respective indices j and k as required by $\textsc{BoundDel}$ also exist for $\textsc{FiniteDel}$.

4. Pick any $\pi_1 \in \Psi$ and any $i \in \mathbb{N}$ such that $\pi_1, i \models \beta$. $\textsc{ExactDel}$ diagnosability states that for any other trace $\pi_2 \in \Psi$ that has the same observations as π_1 up to $i + d$ it must hold that $\pi_2, i \models \beta$. This also holds for any $d' > d$, because we will still have $\pi_2, i \models \beta$, whether $obs(\pi_1^{i+d'}) = obs(\pi_2^{i+d'})$ or not.

5. Similarly, the index j as required by $\textsc{BoundDel}$ trace diagnosability is also a valid choice for any $d' > d$, since the index k as required for d is also a valid choice for d', since we will still have $\pi_2, k \models \beta$ for any π_2 with same observability up to j.

$\square$

4.2 Verification: Algorithms

The verification of diagnosability for an alarm condition Ξ on the plant S, as defined in the previous section, amounts to proving the absence of a critical pair. In this section, we reduce verification of diagnosability for a given alarm specification Ξ to the problem of model-checking a suitable temporal formula on a model that is derived from S, called the *twin plant*. The twin plant construction is based on two copies of S, such that a trace

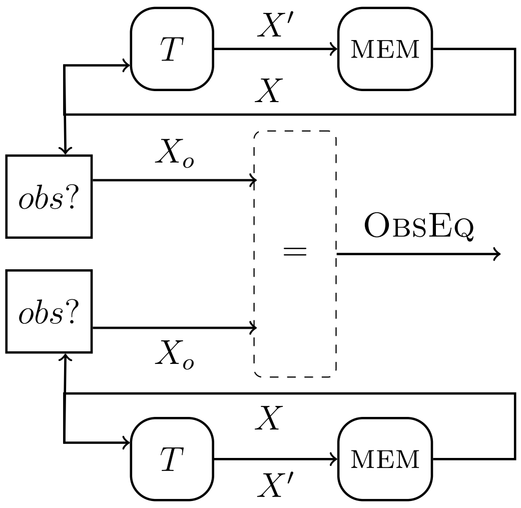


Figure 4.6: Twin Plant

in the twin plant corresponds to a pair of traces of S. The temporal formula $CP(\Xi, \Psi)$ constrains the two traces to be a critical pair for the alarm condition Ξ with respect to Ψ. We show how to define the twin plant as a symbolic transition system, and the temporal property to be checked.

4.2.1 Twin Plant

The idea of *coupled twin plant* for a plant S, originally proposed by [Jiang et al., 2001], is a plant obtained from two copies of S. The state of $\textsc{Twin}(S)$ is composed of the state components of the two copies of the plant, which evolve independently from each other, according to the transition relation T. The output of $\textsc{Twin}(S)$ is a Boolean variable, obtained by comparing the two sets of observable variables of the two copies of S (see Figure 4.6).

Definition 18 (Twin Plant). *The* twin plant *of $S = \langle X, X_o, I, T \rangle$ is the STS* $\textsc{Twin}(S) = \langle VV, \emptyset, II, TT \rangle$*, where:*

- $VV \doteq X_L \cup X_R \cup \textsc{ObsEq}$*;*

- $II(VV) \doteq I(X_L) \wedge I(X_R) \wedge (\textsc{ObsEq} \leftrightarrow \bigwedge_{x \in X_o} x_L = x_R)$
- $TT(VV, VV') \doteq T(X_L, X'_L) \wedge T(X_R, X'_R) \wedge$
  $\textsc{ObsEq}' \leftrightarrow (\textsc{ObsEq} \ \wedge \ \bigwedge_{x \in X_o} x'_L = x'_R)$

The totality of $\textsc{Twin}(S)$ follows from the totality of S. There is a one-to-one correspondence between $\Pi(S) \times \Pi(S)$ (pairs of traces of S) and $\Pi(\textsc{Twin}(S))$ (traces of $\textsc{Twin}(S)$). The variable $\textsc{ObsEq}$ keeps track on whether the observable state variables have diverged in the past. If it evaluates to false, then on the two traces at some point two observables had different values between the two copies. We will use this variable to identify critical pairs, on which we require that the observations do not diverge up to some point of interest.

A trace of $\textsc{Twin}(S)$

$$\pi_L \pi_R \doteq (s_{0,L}, s_{0,R}, \textsc{ObsEq}_0), (i_{1,L}, i_{1,R}), (s_{1,L}, s_{1,R}, \textsc{ObsEq}_1), \ldots$$

can be decomposed into two traces of S:

$$\pi_L \doteq s_{0,L}, i_{1,L}, s_{1,L}, \ldots \qquad \text{and} \qquad \pi_R \doteq s_{0,R}, i_{1,R}, s_{1,R}, \ldots$$

Conversely, given two traces in $\Pi(S)$, the corresponding trace in $\Pi(\textsc{Twin}(S))$ can be reconstructed by simulating the value of $\textsc{ObsEq}$ over time.

Theorem 14. *Let $\pi_L, \pi_R \in \Pi(S)$ be two traces. π_L and π_R are such that $obs(\pi_L^k) = obs(\pi_R^k)$ (i.e., indistinguishable up to k) iff the trace $\pi_L \pi_R \in \Pi(\textsc{Twin}(S))$ is such that $\pi_L \pi_R[k] \models \textsc{ObsEq}$.*

Proof. Using the definition of $\textsc{ObsEq}$ in Definition 18, we proceed by induction on k.

$k = 0$) $obs(\pi_L^0) = obs(\pi_R^0)$ iff $obs(s_{0,L}) = obs(s_{0,R})$ iff $s_{0,R} s_{0,L} \models \bigwedge_{x \in X_o} x_L = x_R$ iff (Definition 18) $s_{0,R} s_{0,L} \models \textsc{ObsEq}$ iff $\pi_L \pi_R[0] \models \textsc{ObsEq}$

Ξ		$CP(\Xi, \Psi)$				
$\textsc{ExactDel}(\circ, \beta, d)$	$\doteq$	$\Psi_L \wedge \Psi_R \wedge \mathsf{F}\ (\textsc{ObsEq}$	$\wedge$	$\mathsf{Y}^d \beta_L$	$\wedge$	$\mathsf{Y}^d \neg\beta_R)$
$\textsc{BoundDel}(\circ, \beta, d)$	$\doteq$	$\Psi_L \wedge \Psi_R \wedge \mathsf{F}\ (\textsc{ObsEq}$	$\wedge$	$\mathsf{Y}^d \beta_L$	$\wedge$	$\mathsf{H}^{\leq 2d} \neg\beta_R)$
$\textsc{FiniteDel}(\circ, \beta)$	$\doteq$	$\Psi_L \wedge \Psi_R \wedge (\mathsf{G}\ \textsc{ObsEq})$	$\wedge$	$(\mathsf{F}\ \beta_L)$	$\wedge$	$(\mathsf{G}\ \neg\beta_R)$

Table 4.1: Critical Pair: LTL formulae

$k > 0$) The inductive hypothesis tells us that the two traces are indistinguishable up to $k-1$ iff $\pi_L \pi_R[k-1] \models \textsc{ObsEq}$. By using this fact, we just need to show that the relation is preserved for the state s_k.

Formally, $obs(\pi_L^k) = obs(\pi_R^k)$ iff $obs(\pi_L^{k-1}) obs(s_{k,L}) obs(i_{k,L}) = obs(\pi_R^{k-1}) obs(s_{k,R}) obs(i_{k,R})$ iff $\pi_L \pi_R[k-1] \models \textsc{ObsEq}$ and $s_{k,R} s_{k,L} \models \bigwedge_{x \in X_o} x_L = x_R$ iff (Definition 18) $\pi_L \pi_R[k] \models \textsc{ObsEq}$

□

4.2.2 Verification via Model-Checking

A result of Theorem 14 is that critical pairs of the plant can be mapped to special traces on the twin plant. Thus, it is possible to reduce the problem of diagnosability for S to a problem of finding those special traces in $\textsc{Twin}(S)$. This problem can then be solved using model-checking techniques. In particular, for each pattern Ξ defined in the previous section, we generate – given the diagnosis condition and the delay – a temporal property $CP(\Xi, \Psi)$ to be checked on $\textsc{Twin}(S)$, as defined in Table 4.1. We call such formula the *critical pair property*, that is satisfied by the critical pairs of Ξ in Ψ.

In the following we assume that the traces in a context can be characterized by an LTL property. With a little abuse of notation, we interchangeably use Ψ as a subset of $\Pi(S)$ and as an LTL formula characterizing that subset. In the formulae of Table 4.1, we write β_L for $\beta(X_L)$ and β_R for

$\beta(X_R)$, and similarly for Ψ_L and Ψ_R. For instance, $\beta_L \wedge \neg\beta_R$ identifies the states of the twin plant where the left state satisfies the diagnosis condition, and the right one does not. Ψ_L forces the subtrace over L symbols of a twin plant trace to belong to Ψ.

Intuitively, the property $CP(\textsc{ExactDel}(\circ,\beta,d),\Psi)$ states the existence of a pair of traces in Ψ that are (i) indistinguishable until a given time t, (ii) d steps before (i.e. $t-d$) β holds in the left trace, and (iii) in the same state $(t-d)$ β does not hold on the right one. For the bounded case, $CP(\textsc{BoundDel}(\circ,\beta,d),\Psi)$ states the existence of a pair of traces in Ψ that are (i) indistinguishable until a given time t, (ii) d steps before (i.e. $t-d$) β holds in the left trace, and (iii) β does not hold on the right trace in the range from $t-2d$ to t. In the finite delay case, the property $CP(\textsc{FiniteDel}(\circ,\beta),\Psi)$ states the existence of a pair of traces in Ψ that are (i) indistinguishable, (ii) β holds at some unspecified point in the left trace, and (iii) β never holds in the right one.

Theorem 15. *Let S be a plant, $\Psi \subseteq \Pi(S)$ a context, and Ξ an alarm condition. A critical pair for Ξ exists in Ψ iff* $\textsc{Twin}(S) \not\models \neg CP(\Xi,\Psi)$.

Proof. We reason by cases and use the LTL semantics to prove the theorem. We use Theorem 14 to guarantee that we only look at pairs of traces that are observationally equivalent up to some $k \in \mathbb{N}$.

Exact Let $\Xi := \textsc{ExactDel}(\circ,\beta,d)$. $\textsc{Twin}(S) \not\models \neg(\Psi_L \wedge \Psi_R \wedge \mathsf{F}\,(\textsc{ObsEq} \wedge \mathsf{Y}^d\beta_L \wedge \mathsf{Y}^d\neg\beta_R))$ iff $\exists\pi\pi \in \Pi(\textsc{Twin}(S))$, decomposable in $\pi_L\pi_R$, such that (by semantics of LTL) $\pi_L \in \Psi$, $\pi_R \in \Psi$, $\exists k \in \mathbb{N}$ with $k \geq d$ such that $\pi\pi, k \models \textsc{ObsEq}$, i.e. $obs(\pi_L^k) = obs(\pi_R^k)$, $\pi_L, k-d \models \beta$, and $\pi_R, k-d \not\models \beta$, iff $\pi_L\pi_R \in \Psi$ are a critical pair for Ξ.

Bounded Let $\Xi := \textsc{BoundDel}(\circ,\beta,d)$. $\textsc{Twin}(S) \not\models \neg(\Psi_L \wedge \Psi_R \wedge \mathsf{F}\,(\textsc{ObsEq} \wedge \mathsf{Y}^d\beta_L \wedge \mathsf{H}^{\leq 2d}\neg\beta_R))$ iff $\exists\pi\pi \in \Pi(\textsc{Twin}(S))$, decomposable in $\pi_L\pi_R$, such that (by semantics of LTL) $\pi_L \in \Psi$, $\pi_R \in \Psi$,

$\exists k \in \mathbb{N}$ with $k \geq d$ such that $\pi\pi, k \models \textsc{ObsEq}$, i.e. $obs(\pi_L^k) = obs(\pi_R^k)$, $\pi_L, k-d \models \beta$, and $\forall j \in [k-2d, k] \cdot \pi_R, j \not\models \beta$, iff $\pi_L\pi_R \in \Psi$ are a critical pair for Ξ.

Finite Let $\Xi := \textsc{FiniteDel}(\circ, \beta)$. $\textsc{Twin}(S) \not\models \neg(\Psi_L \wedge \Psi_R \wedge (\mathsf{G}\ \textsc{ObsEq}) \wedge (\mathsf{F}\ \beta_L) \wedge (\mathsf{G}\ \neg\beta_R))$ iff $\exists \pi\pi \in \Pi(\textsc{Twin}(S))$, decomposable in $\pi_L\pi_R$, such that (by semantics of LTL) $\pi_L \in \Psi$, $\pi_R \in \Psi$, $\forall k \in \mathbb{N} \cdot obs(\pi_L^k) = obs(\pi_R^k)$ and thus $obs(\pi_L) = obs(\pi_R)$, $\pi_L, j \models \beta$ for some $j \in \mathbb{N}$, and $\pi_R, j \not\models \beta$ for any $j \in \mathbb{N}$, iff $\pi_L\pi_R \in \Psi$ are a critical pair for Ξ.

$\square$

Theorem 16 establishes several monotonicity properties of critical pairs w.r.t. different alarm conditions. It is in some sense dual to Theorem 13, which describes monotonicity properties in cases a specification *is* diagnosable. In particular, we show in Theorem 16 that critical pairs for BoundDel are also critical pairs for ExactDel, that critical pairs for FiniteDel are also critical pairs for BoundDel, and that critical pairs for ExactDel and BoundDel w.r.t. a delay d are also witnesses for non-diagnosability with a delay $d' < d$. Cases 3. and 4. have an additional condition that is satisfied by the way we build the twin plant.

Theorem 16 (Monotonicity)**.** *The following conditions hold for all β, d, d'.*

1. $\models CP(\textsc{BoundDel}(\circ, \beta, d), \Psi) \to CP(\textsc{ExactDel}(\circ, \beta, d), \Psi)$
2. $\models CP(\textsc{FiniteDel}(\circ, \beta), \Psi) \to CP(\textsc{BoundDel}(\circ, \beta, d), \Psi)$
3. $\models (\textsc{ObsEq} \to H\textsc{ObsEq}) \to$
   $(CP(\textsc{ExactDel}(\circ, \beta, d'), \Psi) \to CP(\textsc{ExactDel}(\circ, \beta, d), \Psi))$
   (for all $d' > d$)
4. $\models (\textsc{ObsEq} \to H\textsc{ObsEq}) \to$
   $(CP(\textsc{BoundDel}(\circ, \beta, d'), \Psi) \to CP(\textsc{BoundDel}(\circ, \beta, d), \Psi))$
   (for all $d' > d$)

Proof.

1. $\models CP(\textsc{BoundDel}(\circ, \beta, d), \Psi) \rightarrow CP(\textsc{ExactDel}(\circ, \beta, d), \Psi)$
   iff
   $[\Psi_L \wedge \Psi_R \wedge \mathsf{F}\ (\textsc{ObsEq} \wedge \mathsf{Y}^d\beta_L \wedge \mathsf{H}^{\leq 2d}\neg\beta_R)] \rightarrow [\Psi_L \wedge \Psi_R \wedge \mathsf{F}\ (\textsc{ObsEq} \wedge \mathsf{Y}^d\beta_L \wedge \mathsf{Y}^d\neg\beta_R)]$.

   Syntactically the expressions are similar and we just need to show that $\mathsf{H}^{\leq 2d}\neg\beta_R \rightarrow \mathsf{Y}^d\neg\beta_R$. This is true at any point $i >= d$ of any trace. Note that we only need to consider points $i >= d$, because otherwise we get that the left-hand side of the implication is false (due to $Y^d\beta_L$).

2. $\models CP(\textsc{FiniteDel}(\circ, \beta), \Psi) \rightarrow CP(\textsc{BoundDel}(\circ, \beta, d), \Psi)$
   iff
   $[\Psi_L \wedge \Psi_R \wedge (\mathsf{G}\ \textsc{ObsEq}) \wedge (\mathsf{F}\ \beta_L) \wedge (\mathsf{G}\ \neg\beta_R)] \rightarrow [\Psi_L \wedge \Psi_R \wedge \mathsf{F}\ (\textsc{ObsEq} \wedge \mathsf{Y}^d\beta_L \wedge \mathsf{H}^{\leq 2d}\neg\beta_R)]$.

   We consider only traces that satisfy the context, and $\mathsf{G}(\textsc{ObsEq} \wedge \neg\beta_R)$, and s.t. at a point i the trace satisfies β_L ($\mathsf{F}\beta_L$). The same trace satisfies the right-hand side of the implication at the time point $i + d$.

3. for all $d' > d$: $(\textsc{ObsEq} \rightarrow \mathsf{H}\textsc{ObsEq}) \rightarrow ([\Psi_L \wedge \Psi_R \wedge \mathsf{F}\ (\textsc{ObsEq} \wedge \mathsf{Y}^{d'}\beta_L \wedge \mathsf{Y}^{d'}\neg\beta_R)] \rightarrow [\Psi_L \wedge \Psi_R \wedge \mathsf{F}\ (\textsc{ObsEq} \wedge \mathsf{Y}^d\beta_L \wedge \mathsf{Y}^d\neg\beta_R)])$.

   Let us call t the point in the trace that satisfies the eventuality $(\textsc{ObsEq} \wedge \mathsf{Y}^{d'}\beta_L \wedge \mathsf{Y}^{d'}\neg\beta_R)$, and let $k = d' - d$, and $t - k$ be the point in which we evaluate the corresponding eventuality $(\textsc{ObsEq} \wedge \mathsf{Y}^d\beta_L \wedge \mathsf{Y}^d\neg\beta_R)$. It follows that we are evaluating β_L and β_R on the same point $t - d'$, and therefore we only need to show that if $t \models \textsc{ObsEq}$ then $t - k \models \textsc{ObsEq}$. However, this follows from the assumption that $\textsc{ObsEq} \rightarrow \mathsf{H}\ \textsc{ObsEq}$.

4. This follows the same reasoning as case 3.

□

4.3 Synthesis: Problem Definition

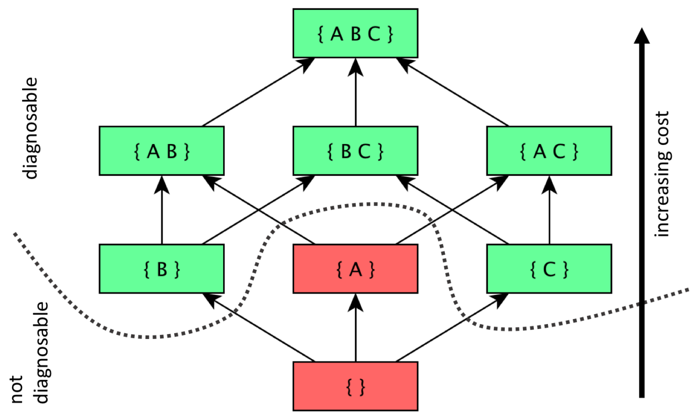


Figure 4.7: Example lattice of sensor configurations.

We now consider the problem of synthesis for diagnosability. The idea is to provide ways to compute a set of configurations of sensors that are sufficient to perform diagnosis. Once we have this set, we can rank the configurations according to various criteria. Specifically, given a set of sensors, we would like to find all sensor configurations (i.e. subsets of sensors) that ensure diagnosability and may also respect some other useful properties (e.g. being optimal with respect to a given cost function). This can provide practical benefits in many domains, by reducing costs, weight, power-consumption, or simply by allowing to replace more complex sensors with simpler ones. A sensor configuration is sometimes referred to as a set of *observability requirements*, since it describes the information that is required to be observable in order for the diagnosis to be possible [Bittner et al., 2012].

In order to study the impact of different sensor configurations on the diagnosability of the plant, we introduce the concept of plant restriction w.r.t. a sensor configuration, in which we consider a modified plant in which only the sensors in the sensor configuration can be used.

Definition 19 (Sensor Configuration, Plant Restriction)**.** *Let $S = \langle X, X_o, I, T\rangle$ be a plant. A* sensor configuration *for S is a subset of X_o. The restriction of S to the sensor configuration sc, written $S_{\downarrow sc}$, is $\langle X, X_o \cap sc, I, T\rangle$.*

We can thus apply the techniques described in Section 4.2 to verify whether a given sensor configuration sc is sufficient to make the alarm condition diagnosable in the system. The set of all such configurations sc is defined as follows.

Definition 20 (DiagSC)**.** *Let an alarm condition Ξ and a context Ψ for the plant S be given. The set of diagnosable sensor configurations, denoted with DiagSC(S, Ξ, Ψ), is the set of sensor configurations $sc \subseteq X_o$ such that Ξ is diagnosable in $S_{\downarrow sc}$ relative to Ψ.*

Figure 4.7 shows an example lattice of sensor configurations for a setting with three sensors: A, B, C. The configurations marked with green (all supersets of $\{B\}$ and $\{C\}$) are diagnosable, corresponding to DiagSC; the configurations marked in red are not diagnosable. Once we have this set DiagSC, we can perform different types of analyses. For example, we might be interested in finding the configurations that are minimal w.r.t. subset inclusion, optimal w.r.t. a cost function, or Pareto-Optimal [Pareto, 1906] w.r.t. multiple cost functions.

The set of *minimal sensor configurations* DiagMinSC(S, Ξ, Ψ) is defined as:

$$\{sc \in \text{DiagSC}(S, \Xi, \Psi) \mid \textit{for all } sc' \subsetneq sc,\ sc' \notin \text{DiagSC}(S, \Xi, \Psi)\}.$$

In Figure 4.7, DiagMinSC corresponds to $\{B\}$ and $\{C\}$. An interesting (and useful) property of diagnosability is its monotonicity w.r.t. to set inclusion.

Theorem 17. *If Ξ is diagnosable in $S_{\downarrow sc}$, then it is diagnosable in $S_{\downarrow sc'}$ for all sc' such that $sc \subseteq sc' \subseteq X_o$; that is, $DiagSC(S, \Xi, \Psi)$ is upward-closed: $sc \in DiagSC(S, \Xi, \Psi)$ implies that for all $sc' \subseteq X_o$ such that $sc' \supseteq sc$ then $sc' \in DiagSC(S, \Xi, \Psi)$.*

Proof. This follows from Definitions 14, 15, and 16. Let $sc' \supset sc$, s.t. $sc' \subseteq X_o$, and let us assume that $sc \in \text{DiagSC}(S, \Xi, \Psi)$. We will show that $sc' \in \text{DiagSC}(S, \Xi, \Psi)$ as well.

In particular, we discuss it for FINITEDEL$(\circ, \beta)$, but the same reasoning can be applied to the other cases. Recall from Definition 16 that Ξ is diagnosable iff $\pi_1, i \models \beta \Rightarrow (\exists j \in \mathbb{N}\ i \leq j \cdot \forall \pi_2 \in \Psi \cdot obs(\pi_1^j) = obs(\pi_2^j) \Rightarrow \exists k \in \mathbb{N}\ k \leq j \cdot \pi_2, k \models \beta)$. If this is satisfied, then increasing the number of observables will preserve its satisfaction. Notice that a change in observable variables only affects the projection function $obs(\cdot)$ of the definitions. By declaring more variables as observable, the projection will contain the same variables as before, plus the new ones. Let π_1, π_2, j be such that they satisfy the above condition. If we increase the number of observables, then either they are not observationally equivalent anymore (and thus the implication becomes vacuously satisfied), or if they are, then π_2 must satisfy the additional conditions (i.e., $\exists k \in \mathbb{N}\ k \leq j \cdot \pi_2, k \models \beta$) since this is not changed by the choice of observables. When quantifying over observationally equivalent traces and increasing the set of observables we can end-up with a set that is a subset (or at most equal) to the previous. Thus, we do not need to consider any additional combination of π_1, π_2, j.

□

This result tells us that knowing DiagMinSC we can obtain also DiagSC. In Figure 4.7 it can be seen how DiagSC consists of all supersets of the minimal configurations $\{B\}$ and $\{C\}$.

As with DiagSC, also with DiagMinSC we can perform interesting analyses. For instance, we might ask what sensor is not present in any minimal configuration; this should give some hints on what sensors give a weak contribution in diagnosing the diagnosis condition at hand. Conversely, we might ask whether there is a sensor that is present in all configurations, and thus essential for diagnosability.

When redundancy of sensors becomes important, the property of minimality might not be sufficient. If any sensor s of a minimal sensor configuration sc is removed, the alarm condition is (by definition of minimality) not diagnosable anymore, and w.r.t. diagnosability we thus have $|sc|$ single points of failure. However, if we have the complete DiagSC, we can easily search for sensor configurations that are robust w.r.t. the outage of any single sensor, by considering only configurations whose immediate predecessors in the lattice are still diagnosable. In Figure 4.7 such a set is the configuration $\{B, C\}$.

Given a cost function of the form $\textsc{Cost} : 2^{X_o} \rightarrow \mathbb{R}$, the set of *cost-optimal sensor configurations* DiagOptSC($S, \Xi, \Psi, \textsc{Cost}$) is defined as:

$$\{sc \in \text{DiagSC}(S, \Xi, \Psi) \mid \textit{for all } sc' \in \text{DiagSC}(S, \Xi, \Psi), \textsc{Cost}(sc) \leq \textsc{Cost}(sc')\}$$

Assume that $\textsc{Cost}$ is monotonic w.r.t. set-inclusion, i.e. $\textsc{Cost}(sc_1) < \textsc{Cost}(sc_2)$ if $sc_1 \subsetneq sc_2$. Intuitively, this means that by adding sensors to a configuration we always increase its cost. By virtue of this property we can limit the search within DiagMinSC instead of working on the whole DiagSC. Note that from DiagOptSC we cannot deduce the whole set of DiagSC. If A, B, and C have costs 1, 2, and 3, respectively, then DiagOptSC consists of $\{B\}$, whose supersets are a strict subset of DiagSC. The computation of DiagOptSC might make sense only when precise information on costs is available, for instance at more advanced project stages. At earlier exploratory stages of a project an engineer might rather focus on performing various what-if analyses based on the complete set of good configurations (DiagSC).

It can also be the case that multiple cost functions are present (e.g. weight, production cost, power-consumption), and they must be taken into account at the same time. Formally, consider M cost functions $\textsc{Cost}_i : 2^{X_o} \to \mathbb{R}$, with $\vec{\textsc{Cost}}(sc) \doteq \langle \textsc{Cost}_1(sc); \ldots; \textsc{Cost}_M(sc)\rangle$. We write $\langle a_1; \ldots; a_M\rangle < \langle b_1; \ldots; b_M\rangle$ iff for all i $a_i \leq b_i$ and for some i $a_i < b_i$. The Pareto front for diagnosability is the set $\text{DiagParSC}(S, \Xi, \Psi, \vec{\textsc{Cost}})$, defined as:

$$\{sc \in \text{DiagSC}(S, \Xi, \Psi) \mid \textit{for all } sc' \in \text{DiagSC}(S, \Xi, \Psi), \vec{\textsc{Cost}}(sc') \not< \vec{\textsc{Cost}}(sc)\}$$

If the cost function is monotonic, then every (Pareto) optimal configuration is also minimal: i.e., $\text{DiagOptSC}(S, \Xi, \Psi, \textsc{Cost}) \subseteq \text{DiagMinSC}(S, \Xi, \Psi)$ and $\text{DiagParSC}(S, \Xi, \Psi, \vec{\textsc{Cost}}) \subseteq \text{DiagMinSC}(S, \Xi, \Psi)$.

Finally, note how the notion of Pareto-optimality is a generalization not only of optimality w.r.t. mono-dimensional cost functions, but also w.r.t. sensor configuration minimality. The latter can be seen as having one cost function per sensor, where each cost function returns a high cost when the sensor is activated, and a low cost when it is not. In the following we will focus on the two optimization problems commonly studied in the literature, i.e. identifying DiagOptSC and DiagMinSC, and explore different strategies for computing them.

4.4 Synthesis: Algorithms

Computing sensor configurations under which a given alarm condition is diagnosable can be achieved in various ways. In this section we first describe various enumerative approaches from the literature adapted to our setting (Section 4.4.1), and then propose a fully symbolic approach to manage the search (Section 4.4.2). Whereas the enumerative approaches try to directly compute DiagMinSC and DiagOptSC, the symbolic approach first computes DiagSC and applies any optimization criteria afterwards. Furthermore, while the enumerative methods exploit certain diagnosability-specific characteristics of the problem, the symbolic approach is based on a general parameter synthesis technique for monotonic problems. We briefly mention the possible extension of parameter synthesis with costs in Section 4.4.3 which could be an interesting way to efficiently compute sensor configurations, but is an open research problem. Finally, we describe a schematic overview of problems and algorithmic approaches to synthesis, pointing out some interesting directions for future work (Section 4.4.4).

Due to the incompleteness of the critical-pair approach in the cases mentioned in Section 4.1.2 we obtain over-approximations of the sets that we would actually want to compute. For BoundDel, in case the diagnosis condition is not monotonic and when synthesizing for a delay d we obtain solutions that are guaranteed to be diagnosable with a delay of $2d$; the true bound for individual solutions might of course be lower. For FiniteDel, in case of infinite-state models or liveness properties defining the context, the set of configurations being computed may contain elements that are not diagnosable at all, which limits its applicability. The returned set however can still be useful, as it gives information on what observables are necessary (though not sufficient) to guarantee diagnosability.

4.4.1 Synthesis by Enumeration

The simplest way to perform synthesis is to enumerate all sensor configurations and verify for each whether it makes the alarm condition diagnosable. This approach, however, has a complexity that is exponential in the number of sensors. Some algorithms in the literature aim to find just one solution. To find one configuration in DiagMinSC is indeed linear in the number of sensors; one just needs to try dropping the sensors one-by-one, and keep-

```
function ComputeDiagOptSC ($S, \Xi, \Psi, \textsc{Cost}$)
1   SC := $\{2^{X_O}\}$
2   do
3     $sc := argmin_{sc \in SC}(\textsc{Cost}(sc))$
4     $\pi := check[\ \textsc{Twin}(S_{\downarrow sc}) \models \neg CP(\Psi, \Xi)\ ]$
5     if $(\pi = nil)$ return $sc$
6     $SC := \{sc \in SC | sc \cap getObsReq(S, \pi, sc) \neq \emptyset\}$
7   while $SC \neq \emptyset$
8   return $nil$ // Non-Diagnosable
```

Figure 4.8: Algorithm to compute one element of DiagOptSC.

ing only those where a critical pair is generated when they are dropped (see the top-down approaches in [Jiang et al., 2003b, Briones et al., 2008]). However, already looking for a single solution with minimum cardinality (every sensor has a cost of 1) is, for finite-state automata, NP-complete as shown in [Yoo and Lafortune, 2002a].

To lower the impact of the problem's complexity, in [Briones et al., 2008] a technique is introduced that allows a more informed search, exploiting domain-specific properties. It is also enumerative in nature, but analyzes any given critical pair and identifies what specific observables can distinguish it. Specifically, it tries to identify what individual sensors can be activated such that the corresponding observable projection of the two traces in the critical pair becomes different. This technique is combined with the bottom-up strategy of [Jiang et al., 2003b], an algorithm that identifies one element of DiagMinSC by incrementally adding observables.

Based on this idea of analyzing critical pairs, in [Grastien, 2009] a best-first approach using a cost function is proposed to guide the enumeration of the sensor configurations, with the goal of identifying one element of DiagOptSC. The outline of this algorithm is shown in Figure 4.8.

The approach is bottom-up, following the monotonicity of the cost function. At each iteration, we pick the next best configuration *sc* according to the cost function COST (Line 3), and then perform the diagnosability check (Line 4). The first sensor configuration to be checked is the "empty" one (i.e. no sensor at all is used), which has a cost of 0. If no critical pair is found, *sc* is sufficient to make the system diagnosable (within the fragments identified in Section 4.1), and since we performed a best-first search, we know that it is also cost-optimal, i.e. a member of DiagOptSC, and we return it (Line 5). Otherwise, we call the function $\mathit{getObsReq}(S, \pi, sc)$ (Equation 4.1) to understand what sensors can distinguish it and remove all configurations from the search space that do not have at least one such sensor (Line 6). Whenever a critical pair is found that cannot be disambiguated even by using all possible sensors, $\mathit{getObsReq}(S, \pi, sc)$ will evaluate to the empty-set, SC will be set to $\emptyset$, and the algorithm will return "not-diagnosable" (Line 8).

Note that we can generalize this best-first search procedure and optimize for set-inclusion, thus looking for an element of DiagMinSC. For this we have to pick, in Line 3, a candidate *sc* that is not a superset of any other candidate sc'. If it turns out to be diagnosable, it is guaranteed to be a member of DiagMinSC.

The set SC has two loop-invariant properties which are guaranteed by the update in Line 6. First, it contains only sensor configurations that have not been explored yet (i.e. it cannot contain a set *sc* that was checked in some previous iteration); second, it contains only elements that are hitting sets over all observability requirements identified by *getObsReq*() so far (guaranteed by the intersection test).

The function $\mathit{getObsReq}(S, \pi, sc)$ inspects a trace (π) of $\textsc{Twin}(S_{\downarrow sc})$, and returns the set of sensors that are sufficient to disambiguate the critical pair π:

$$\{s \in (X_o \setminus sc) \mid \textsc{Twin}(S_{\downarrow sc \cup \{s\}}), \pi_L \pi_R \not\models CP(\Xi, \Psi)\} \tag{4.1}$$

As pointed out in [Briones et al., 2008], each of the new sensors is sufficient to disambiguate the given trace. However, note that here the notion of *getObsReq* is more subtle because of the increased expressiveness of the patterns being analyzed. In our framework it is not sufficient to choose a sensor that will cause the observations to be *eventually* different, but we need the sensor to be able to disambiguate the traces within the time-bound imposed by the delay of the alarm configuration. For instance, for an alarm condition $\textsc{BoundDel}(\circ, \beta, 5)$, if β occurs at time t, then we need the new sensor to disambiguate the traces within $t + 5$ steps.

The termination of the algorithm is guaranteed by the fact that the search space is finite and that at each iteration at least one element of SC is dropped. If we analyzed all sensor configurations without ever finding a diagnosable one, then the system is not diagnosable.

The above algorithm stops after one solution is found; however, we can store this solution and continue the search with the next-best candidate. Due to the monotonicity of the diagnosability problem, we can exclude candidates that are a super-set of previous solutions, thus substantially pruning the search space. This procedure allows us, in principle, to compute the exact sets DiagOptSC and DiagMinSC. As shown in Section 4.5, this can work well in some cases, but in many others it does not scale up, especially for computing DiagMinSC. For this reason, we propose a technique to be able to symbolically reason on the set of sensor configurations without the need of enumerating them one by one.

4.4.2 Synthesis via Parameter Synthesis

The key idea to effectively compute DiagSC is the use of a purely symbolic representation of the state-space of all possible sensor configurations. To this aim we introduce the *Parameterized Twin Plant*. Using this type of twin plant we can reduce the problem of computing DiagSC to a problem of parameter synthesis for model-checking, for which we can apply off-the-shelf tools and algorithms.

Parameterized Twin Plant We first construct a symbolic representation of the set $\{S_{\downarrow sc} | sc \subseteq X_o\}$, i.e., the set of all possible restrictions of S.

Definition 21 (Parameterized Twin Plant)**.** *The* Parameterized Twin Plant *of* $S = \langle X, X_o, I, T \rangle$ *is the symbolic transition system* $\textsc{ParTwin}(S) = \langle VV, \emptyset, II, TT \rangle$*, where:*

- $VV \doteq X_L \cup X_R \cup \textsc{ObsEq} \cup A$*, where* $A = \{\mathrm{A}_s | s \in X_o\}$ *is a set of Boolean variables;*
- $II(VV) \doteq I(X_L) \wedge I(X_R) \wedge (\textsc{ObsEq} \leftrightarrow \bigwedge_{x \in X_o} \mathrm{A}_x \rightarrow x_L = x_R)$
- $TT(VV, VV') \doteq T(X_L, X'_L) \wedge T(X_R, X'_R) \wedge \textsc{NoChange}(A) \wedge \textsc{ObsEq}' \leftrightarrow (\textsc{ObsEq} \wedge \bigwedge_{x \in X_o} \mathrm{A}_x \rightarrow x'_L = x'_R)$,

  where

$$\textsc{NoChange}(A) \doteq \bigwedge_{\mathrm{A}_s \in A} \mathrm{A}'_s \leftrightarrow \mathrm{A}_s$$

The Parameterized Twin Plant is a symbolic representation of the set of all the restrictions of plant S to some sc, based on the values to the so-called *activation parameters* in A. The conjunct $\textsc{NoChange}(A)$ forces the variables in A not to change. The key difference with respect to $\textsc{Twin}(S)$ is the definition of $\textsc{ObsEq}$, that is now conditioned by the value of the activation parameters A. When a parameter is false, the value of the corresponding variable no longer contributes to the observational equivalence, making it (de facto) unobservable. There exists a bijection between sensor configurations and assignments to the activation parameters. Specifically, sc corresponds to an assignment μ_{sc} to the variables in A, such that, for all $s \in X_o$, $\mu_{sc} \models \mathrm{A}_s$ iff $s \in sc$. Instantiating $\textsc{ParTwin}(S)$ with respect to

μ_{sc} results exactly in the $\textsc{Twin}(S_{\downarrow sc})$, i.e. $\textsc{ParTwin}(S)_{\downarrow \mu_{sc}} = \textsc{Twin}(S_{\downarrow sc})$. When $sc = X_o$, and μ_{sc} assigns $\top$ to all the variables in A, we obtain $\textsc{Twin}(S)$. We also notice that the space of configurations induces a partition on the set $\Pi(\textsc{ParTwin}(S))$. Any two partitions contain the same set of traces, modulo projection of the values to A and $\textsc{ObsEq}$.

The symbolic representation results in some clear advantages. First of all, the parametric model allows for an off-the-shelf reuse of routines for parameter synthesis, which, as shown in Section 4.5, has important performance benefits. Second, since subsumption between sensor configurations boils down to logical entailment, it is possible to exploit the monotonicity property. Excluding configurations from the search process can be done symbolically by strengthening the transition relation. For instance, to exclude all configurations that have neither sensor s_1 nor sensor s_2, we can use $TT := TT \wedge (\textsc{a}_{s_1} \vee \textsc{a}_{s_2})$. When looking for DiagSC this means that we do not need to maintain a dedicated data structure to manage the search. Third, the symbolic representation allows to reason over multiple sensor configurations at the same time, instead of looking at a single configuration at a time.

Parameter Synthesis for Model-Checking Based on the parameterized twin plant, we reduce the problem of synthesis for diagnosability to a problem of parameter synthesis for model-checking, for which several algorithms are available (e.g. [Cimatti et al., 2008, André et al., 2009, Cimatti et al., 2013a, Bittner et al., 2014a, Bozzano et al., 2015d]). Due to the monotonicity assumption we can exploit specialized algorithms such as the ones in [Bozzano et al., 2015d] that are used, for instance, in the field of Model-Based Safety Assessment to construct Fault Trees [Bozzano et al., 2007, 2015a].

The parameter synthesis problem is such that we can directly express

```
function ParamSynth ($S, \text{A}, \phi$)
1   $badConfigs := \bot$;
2   do
3     $\pi := check[\ S \models \neg badConfigs \rightarrow \phi\ ]$
4     if ($\pi = \emptyset$) do
5       return ($\neg badConfigs$)
6     endif
7     $curBad := generalize(Proj(A, \pi^0))$
8     $badConfigs := badConfigs \vee curBad$
9   while (true)
```

Figure 4.9: Basic algorithm for symbolic computation of DiagSC via parameter synthesis. S is the parametric system (parameterized twin plant), A is the set of parameters (activation variables), ϕ is the proof obligation any valid solution needs to satisfy ($\neg CP(\Xi, \Psi)$).

the problem:

$$\{sc \mid \textsc{Twin}(S_{\downarrow sc}) \models \neg CP(\Xi, \Psi)\}$$

and obtain the set DiagSC as a result. This is achieved without the need of considering each possible configuration independently.

The algorithm in Figure 4.9 gives a high-level view of the general approach. The idea is to collect all the sets of sensors for which $S_{\downarrow sc}$ is not diagnosable ($badConfigs$ in Line 1), and then complement the set (Line 5). The call to the model-checker looks for a violation of ϕ, in this case a critical pair, caused by one of the unexplored configurations (Line 3). If no violation is found, then all the sensor configurations for which the check fails have been excluded, and the (symbolic representation of the) complement set is returned (Line 5). Otherwise, the bad sensor configuration is extracted from the counterexample, generalized by assumption of monotonicity to a set of bad configurations (Line 7), and added to the set to be blocked (Line 8).

The specific algorithm we use in the implementation (see [Bozzano et al., 2015d]) improves on this algorithm by using the bad configurations to directly strengthen the transition relation, instead of integrating them into the proof obligation. Furthermore it efficiently exploits the monotonicity in the lattice by exploring it via cardinality constraints.

The approach is close in spirit to the *inverse method* [Cimatti et al., 2008, André et al., 2009]. The main difference is in the fact that, because of monotonicity, it is sufficient to block bad configurations using the clause obtained by dropping the negated literals.

Identifying DiagMinSC and DiagOptSC The result of the parameter synthesis call is a symbolic formula over the variables in A representing the set DiagSC. Various symbolic reasoning methods can be used to compute the sets of interest. A simple yet efficient method to extract DiagMinSC is to use a SAT solver to enumerate all minimal models of the returned expression; we chose this method for our implementation described in Section 4.5. Based on this list, and assuming monotonic costs, DiagOptSC can be obtained by filtering all minimal solutions that do not have optimal cost. Other approaches to directly compute cost-optimal solutions in combinatorial problems are for instance discussed in [Bjørner et al., 2015].

4.4.3 Cost-Driven Parameter Synthesis

A third possible approach is the extension of parameter synthesis with a notion of costs, and use this to directly compute DiagMinSC, DiagOptSC, or also DiagParSC, instead of computing the complete set of DiagSC.

Cost-driven parameter synthesis is an open research problem that has not received much attention so far. In [Bittner et al., 2014a] we investigated Pareto-optimal parameter synthesis with two-dimensional cost functions. We consider a generic parameter synthesis problem with Boolean parameters and an invariant property as a proof obligation. The parameter valuations are monotonic w.r.t. property satisfaction: if the proof obligation holds for one particular valuation, it holds for all of its supersets in terms of parameters being set to "true". Furthermore a two-dimensional cost function is assumed as given, which too is monotonic: supersets of valuations have a higher cost on both cost dimensions. The synthesis goal is to find the Pareto-optimal parameter valuations.

The monotonicity assumptions apply also for diagnosability: property satisfaction is upward monotonic w.r.t. the addition of new sensors, and monotonicity of cost functions is also common in practice. As only a very restricted subset of our diagnosability definitions can be mapped to this parameter synthesis framework, here we report only the main findings and discuss possible future work to support a wider set of diagnosability problems.

The approach explores the lattice induced by the cost functions in an enumerative way top-down, i.e. starting from the most expensive valuation. The cost dimensions are optimized in alternation, by fixing one dimension and trying to find the lowest possible cost on the other dimension. The monotonicity on costs and on the parameters are used to prune the search space.

The implementation is based on a parameterized transition system similar to the parameterized twin-plant, representing in a symbolic way all possible instantiations of the "ground" transition system. The choice of parameter assignments is moved inside the proof obligation, and the model-checker can always reason on the same system. This property, the fact that we proceed top-down and that we have invariant proof obligations allows us to exploit an essential feature in the IC3-based implementation. When a property holds, IC3 will return an inductive invariant, represented symbolically, that is an over-approximation of the reachable states, in our case of the parameterized transition system. This inductive invariant can be directly used with a SAT solver to check for property violations of cheaper candidates, instead of calling the model-checker for verifying if a candidate is a valid solution. If it doesn't admit a violation of the proof obligation, then the selected valuation is indeed valid as the check was done with an over-approximation of the reachable states. Else, we switch back to standard model-checking calls.

Part of the experiments include ExactDel diagnosability checks with an empty context (system-diagnosability) and a propositional diagnosis condition. The corresponding proof obligation is reduced to an invariant by adding a counter to the transition system that measures the delay up to divergence of the observable signals. The proposed technique is compared to the standard inverse method, which enumerates bad valuations and prunes the search space using the monotonicity of the valuations while ignoring costs. The performance gain is in orders of magnitude, which clearly shows the potential of the approach. Note that this technique cannot be applied to the "bottom-up" best-first approach of [Grastien, 2009], which is falsification-focused; it tries to first identify non-diagnosable solutions until it hits diagnosable ones, and thus counterexample traces instead of inductive invariants are returned by the model-checker.

To support all diagnosability checks presented in this chapter we will try to extend the approach in future work to generic LTL properties. LTL reasoning using IC3 is an active research area. Here we rely on inductive invariants in order to quickly prune the search space, but a similar concept is not present for many IC3 based LTL algorithms. This would need to be identified and implemented before we can apply the developed parameter synthesis approach to our diagnosability problems.

		DiagSC	DiagMinSC	DiagOptSC	DiagParSC
Cost-Driven Enumerative		Section 4.4.1			
Symbolic	Param-Synthesis	Section 4.4.2			
	Cost-Driven Param-Synthesis	Section 4.4.3			

Figure 4.10: Synthesis of all configurations: Schematic view

4.4.4 Schematic Overview

In this section we discussed several possibilities to solve the synthesis problem. The choice of algorithm depends on the type of problem that needs to be addressed and on whether we want to use symbolic or enumerative techniques. Figure 4.10 provides a schematic overview of the dimensions that we take into account in this chapter. Cost-driven parameter synthesis is still mostly an open research problem. For this reason, in the next section, we focus the experimental evaluation on the cost-driven enumerative approach and on the parameter-synthesis one.

4.5 Experimental Evaluation

This section describes the experiments performed to evaluate the feasibility of the proposed techniques in solving the verification and synthesis problems. We first describe the implementation of the algorithms, then the system models used for the tests, the experimental set-up, and finally the results.

4.5.1 Implementation

For verification of diagnosability we use the LTL model-checking procedures provided by nuXmv, a symbolic model-checker for infinite-state transition systems [Cavada et al., 2014]. The synthesis algorithms proposed in previous sections have been implemented within xSAP [Bittner et al., 2016a], a platform for Model-Based Safety Assessment (MBSA), which relies on nuXmv as a back-end. We use MathSAT [Cimatti et al., 2013b] for the SAT-solving steps. The implementation is also at the core of the diagnosability functionalities of the COMPASS toolset [Bozzano et al., 2009, COMPASS, 2016], a design environment for system-software co-engineering. We implemented three synthesis algorithms: Symbolic, Enumerative-Minimal, and Enumerative-Optimal. All implementations work on the parameterized twin-plant. The enumerative algorithms enforce the choice of a sensor configuration through the proof obligation.

The Symbolic algorithm computes DiagMinSC by first computing DiagSC and then minimizing the result. To compute DiagSC it maps the diagnosability problem to the parameter synthesis framework of nuXmv and calls nuXmv's procedures off-the-shelf. Specifically, the procedure we call implements the approach of [Bozzano et al., 2015d], which explores the lattice of parameter valuations using cardinality layers. In the case of diagnosability this means exploring the lattice of sensor configurations top-down, i.e. starting with all sensors, then exploring all candidates with one sensor less, and so on. As soon as a non-diagnosable configuration is found in such a cardinality layer it is dropped from the search space, along with all its subsets. The synthesis procedure returns a symbolic representation of DiagSC. We use the SAT solver to enumerate all minimal solutions, along with their costs.

The Enumerative-Minimal algorithm tries to directly compute DiagMinSC using the enumerative best-first approach with sensor configuration minimality as an optimization criteria. At each iteration we use a SAT solver to compute a minimal hitting-set *sc* over the sets of sensors obtained by *getObsReq*() for all critical pairs found so far, excluding any solutions that have already been identified along with, by monotonicity, their supersets. We remark that *sc* must be a *hitting* set, because anything else has already been proven not to be diagnosable. Furthermore we require it to be a *minimal* hitting-set, else it could happen that we obtain a non-minimal solution. If the candidate is diagnosable it is added to the set of minimal solutions. If not, *getObsReq*() is executed on the critical pair, the disambiguating observables stored in a symbolic representation, and the next iteration is started. The algorithm continues until the solutions identified so far represent all minimal hitting-sets.

The Enumerative-Optimal algorithm is similar to Enumerative-Minimal, but uses as an optimization criteria the sum of costs of each individual sensor and aims thus at directly computing DiagOptSC. The implementation uses a priority-queue to which candidates are pushed on-demand. At each round a candidate sc with lowest cost is retrieved from the queue and it is checked whether it contains at least one sensor needed to disambiguate all critical pairs found so far, i.e. whether it is a hitting-set of all observational requirements. If yes, diagnosability is checked for sc. If no, then for each disambiguating observable o of the critical pair not covered by sc, a new configuration $sc \cup o$ is added to the queue, which is then polled again. If the diagnosability check fails, the priority queue is updated the same way, adding one new $sc' = sc \cup o$ for each o that can disambiguate the new critical pair. The algorithm iterates until the queue contains no more candidates with optimal cost.

For the two enumerative algorithms we implemented the *getObsReq*() function by parsing the critical pair and collecting all observables that diverge in at least one point of the trace. For BoundDel and ExactDel we furthermore check whether the additional observables disambiguate the two traces “in time”, by asserting that the critical pair at hand is not a critical pair for the chosen alarm condition and the augmented sensor configuration. This is done by asking the model-checker whether the pair of traces is also a critical pair for one of the new sensor configurations.

4.5.2 Benchmark Set

To investigate the performance of diagnosability verification and synthesis with the described algorithms we used the following benchmark models. Some are shared with the experimental evaluation of the TFPG algorithms, but due to the underlying framework of diagnosability we interpret time in them differently. The context definitions, if any, are all expressed by safety properties. The diagnosis conditions are mostly monotonic (permanent faults), with exception of models Orbiter, RoverSmall, RoverBig and wbs.

acex and autogen are directly taken from the TFPG benchmarks. The goal is to diagnose whether certain states have been visited, based on partial observability of the state vector.

Orbiter, RoverSmall, and RoverBig are models of an orbiter and of a planetary rover developed in the OMCARE project [Bozzano et al., 2008, 2011]. These models are taken from the benchmarks in [Bittner et al., 2012]. They describe the functional level, with various relevant subsystems including sensors and failure modes. The diagnosis property used for the benchmarks is whether a component has failed.

CassiniDiscrete, CassiniDiscrete2, and CassiniInfstate are variants of the Cassini model in the TFPG chapter. CassiniInfstate is an infinite-state variant, using reals to describe propellant levels in tanks and flow magnitudes in the pipes. Available observations are commands, spacecraft acceleration, and flow magnitude in various pipe segments. The diagnosis condition is whether the helium tank has a leak. A context is used to limit the system to valve configurations of interest.

c432 is a Boolean circuit used as a benchmark in the DX Competition [Feldman et al., 2010], whose gates can permanently fail (inverted output). The observables are the inputs and output values for the gates of the circuit. The property is whether a single gate of a certain group of gates is faulty. A context specification is used to limit the analysis to single fault cases.

GUIDANCE is also taken from the TFPG chapter. Certain actions and intermediate milestones are the observables. The diagnosis condition is passing a specific state of the procedure.

POWERDIST is taken from the TFPG chapter as well. Commands to the system, power availability in various parts and broken-status of switches are the observables. The diagnosis condition here is the failure of a specific power line; we use a context to avoid triggering of masking failures.

The WBS and X34 use cases are reused as well. Contexts are used to exclude masking faults and to model assumptions on the external controller.

Basic indicators on the size of the models are given in Table 4.2: number of Boolean variables, number of real-valued variables, number of reachable states (where it was feasible to compute them), diameter of the state space (minimum number of steps to reach any reachable state starting from any initial state), and number of observables. Note that these metrics refer to the original plant, not to the (parameterized) twin plant. The complete experiments are available at `http://es.fbk.eu/people/bittner/phd_diag_expeval.tar.bz2`.

4.5.3 Experimental Set-Up

With the experimental evaluation we want to investigate feasibility, scalability w.r.t. alarm patterns, delay bounds, and number of observables, and specifically for synthesis, comparison of symbolic and enumerative approaches to synthesis. To this aim we created several instances for each use case, with variations in alarm pattern, delay bound, and sets of observables, resulting in a total of 589 instances. As especially for synthesis we want to focus on cases that admit at least some solutions, all these

model	# bool var	# real var	# reach	diam	# obs
acex	31	0	$2^{19.4}$	96	21
autogen	99	0	$2^{12.0}$	20	20
guidance	98	0	$2^{47.5}$	70	62
powerdist	83	0	$2^{11.0}$	31	41
c432	356	0	N.A.	N.A.	196
orbiter	39	0	$2^{18.9}$	33	15
roversmall	60	0	$2^{45.9}$	31	20
roverbig	158	0	N.A.	N.A.	62
cassini	176	0	$2^{44.2}$	8	58
cassini2	265	0	$2^{13.3}$	18	56
cassini-inf	122	30	N.A.	N.A.	96
x34	549	0	N.A.	N.A.	491
wbs	1179	0	N.A.	N.A.	167

Table 4.2: Model Properties

instances are diagnosable.

For the experiments we used a cluster where each node has 100GB of RAM and a 12-core CPU running at 2.67GHz. Each test was run on a single core, with a new invocation of the model-checker. Timeout was set at 900 seconds, memory cap to 16GB. Parallel test executions were limited to three to limit time skew.

4.5.4 Results: Verification

The results of the benchmarks show the feasibility of performing verification of diagnosability via symbolic LTL model-checking. Only 22 out of 589 use cases timed out, all belonging to RoverBig, especially with BoundDel and ExactDel. In all other cases the model-checker was able to prove diagnosability.

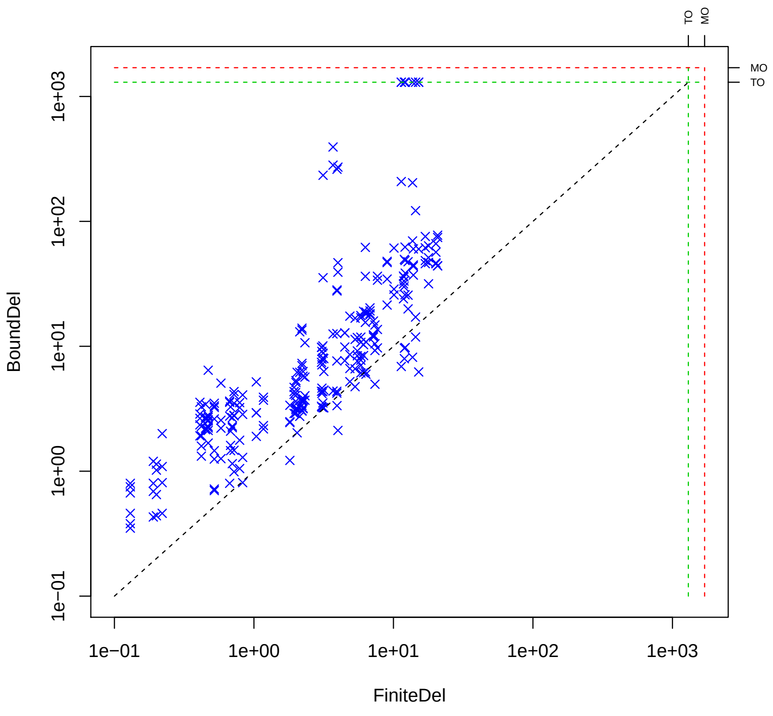
A clear increase in runtime can be observed when using more restrictive alarm patterns. In Figure 4.11a the run-times of FiniteDel instances are compared to all BoundDel instances with same characteristics (model, diagnosis condition, and observables) except bound; for BoundDel vs ExactDel the situation is very similar, and a considerable increase in difficulty can be observed for each case. As a consequence one might need to trade-off verification runtime versus guarantees on diagnosability. For instance, in rapid prototyping sessions response time is usually more important, and one could choose to verify FiniteDel at first, considering the delay bound only later on during the design process.

A similar situation can be observed when increasing the delay bound. This has a recognizable effect on verification runtime in most models, especially for ExactDel. To study these trends in detail we selected two models, one with a low increase in runtime, one with a higher one, and created additional test instances by increasing the delay bound from 0 to 30, by increments of 1. The results are shown in Figure 4.11b and reflect the situation for the other benchmarks. Interestingly, there is no clear change when going from non-diagnosable to diagnosable; *cassini_inf* becomes diagnosable at bound 3 (both patterns); *cassini_dis2* becomes diagnosable at bound 19 for BoundDel, and is not diagnosable with any bound in ExactDel. This result shows that proving diagnosability for a lower bound is potentially easier, and that thus, by monotonicity of diagnosability w.r.t. the delay bound, a strategy based on iterative deepening might pay off for proving diagnosability with a higher bound.

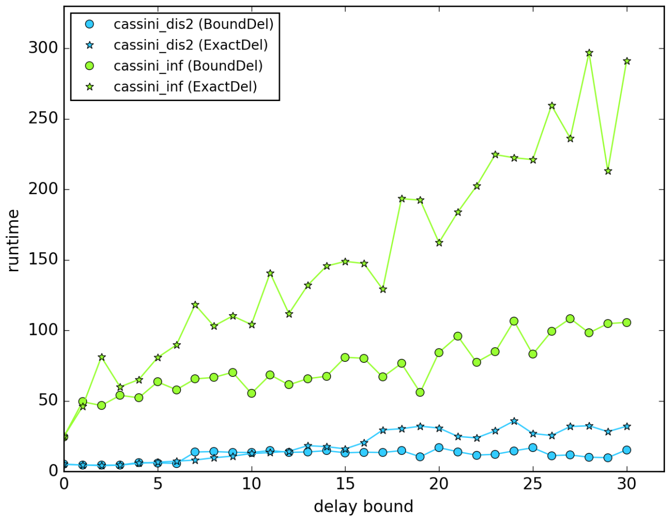
Finally, no correlative pattern between number of observables and verification runtime emerged from the tests.

4.5.5 Results: Synthesis

Synthesis of observables is as expected more difficult, but still feasible, also on the industrial models. The results show the feasibility and in some regards advantage of using an off-the-self implementation of parameter



(a) FiniteDel vs BoundDel



(b) Effect of increasing delay bound.

Figure 4.11: Verification run-times with different patterns and delay bounds.

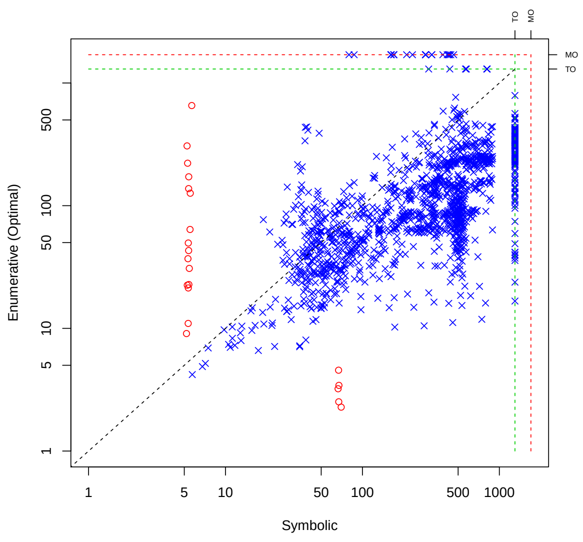
synthesis as opposed to a custom algorithm for synthesis of observables. They also show the relevance of the cost function for the enumerative approach.

As a preliminary remark, note that in the results for Symbolic, the time to extract the minimal solutions from the expression returned by the parameter synthesis engine is for all runs less than $0.2s$.

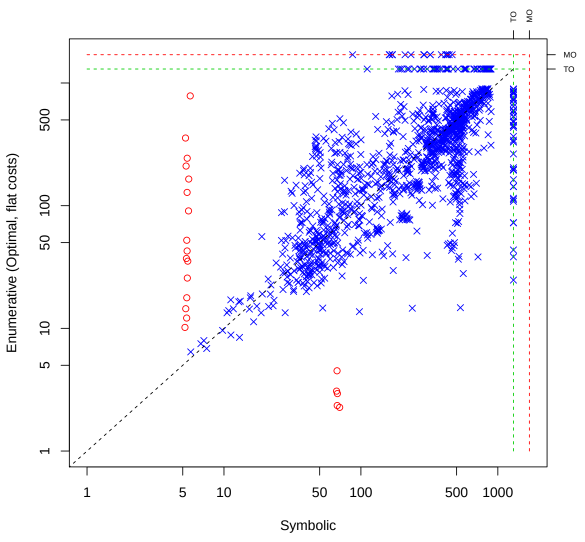
Significant performance differences can be observed between the symbolic method and the enumerative algorithms. When the goal is to compute DiagOptSC, one can use Symbolic and optimize afterwards, or use Enumerative-Optimal to compute the solutions directly. The performance of the enumerative approach as compared to the symbolic one however greatly depends on the specific costs assigned to the sensors.

For the instances in Figure 4.12a we fixed for every observable of each model a random cost between 1 and 8. In this case the enumerative method clearly outperforms the symbolic one, because the search is quickly driven towards the optimal solutions and the cost bound avoids further exploration of the search space.

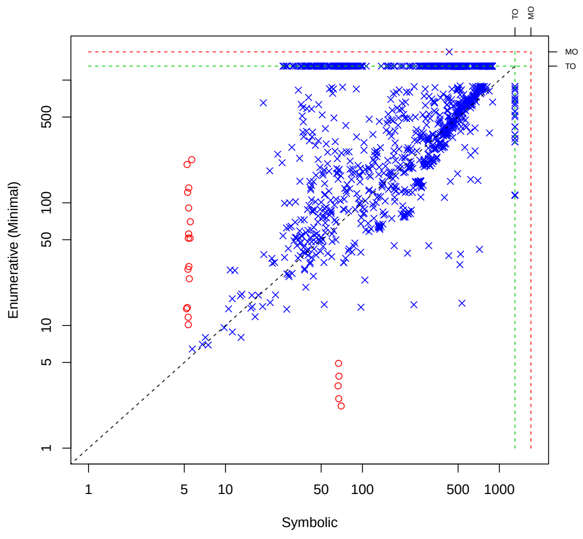
However, when using a flat cost for all sensors (every sensor has cost 1), the performance of Enumerative-Optimal is comparable to Symbolic (see Figures 4.12b and 4.12d), despite the fact that Symbolic also computes all minimal solutions and thus potentially provides a more useful result to the engineer. Flat costs are used when the goal is to optimize the cardinality of the sensor configurations, that is to compute the configurations that use the least number of sensors. This is the case, for instance, when each sensor has the same weight or power consumption, and the goal is to minimize overall weight or power consumption. For synthesis behavior, flat costs mean that the enumerative algorithm, in the worst case, enumerates all candidates in each cardinality layer, up to the layer where the first solution is found. This can work well in cases where solutions have



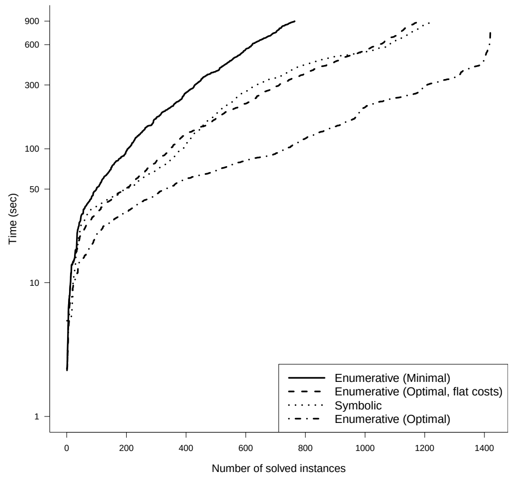
(a) Symb vs. Enum-Optimal



(b) Symb vs. Enum-Optimal (flat costs)



(c) Symb vs. Enum-Minimal



(d) Comparison by solved instances

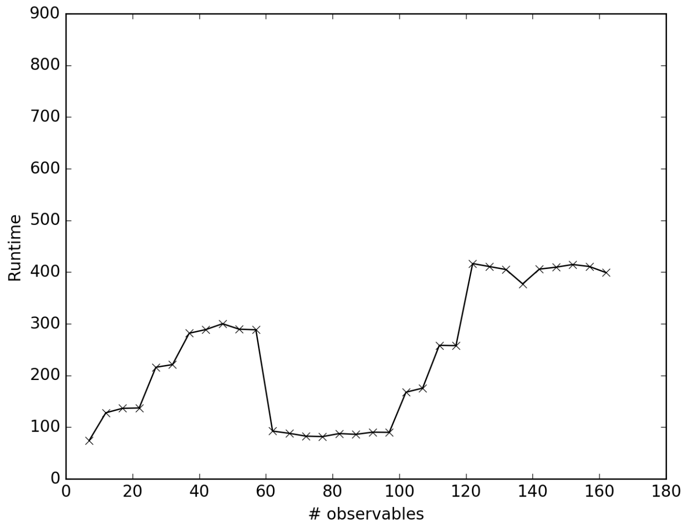
Figure 4.12: Comparison of run-times (seconds) for symbolic vs. enumerative synthesis (TO: out of time, MO: out of memory; blue: diagnosable instances; red: non-diagnosable instances). Total number of instances: 1504; Symbolic solved: 1224; Enumerative-Optimal solved: 1421; Enumerative-Optimal (flat costs) solved: 1181; Enumerative-Minimal solved: 764.

very low cardinality, but quickly degrades when the first solutions are only in higher layers. Indeed, the use-case CassiniDiscrete2 has optimal-cost solutions very low in the lattice (cardinality 1) and the algorithm converges quickly after only a few iterations. On the other hand, the use-case c432 has only one solution, but of cardinality 15, and needs significantly more iterations to converge, if terminating at all.

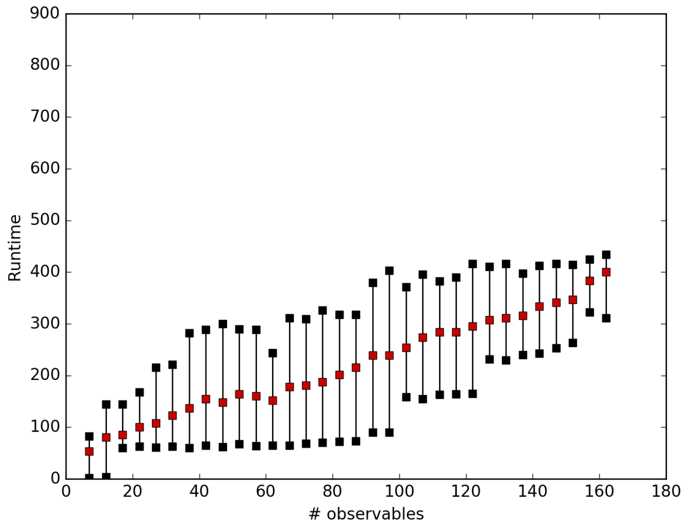
Furthermore, Figures 4.12c and 4.12d show that the symbolic approach is better suited at computing all minimal solutions. Profiling of the algorithms' behavior shows that the symbolic method often needs to identify much fewer critical pairs for convergence. The reason seems to be that through the top-down exploration guided by cardinality layers, the symbolic method identifies "tighter" critical pairs, where fewer observables diverge along the traces, resulting in a higher pruning effect through monotonicity of diagnosability.

In terms of how the alarm pattern affects synthesis runtime, all three algorithms show an increase in difficulty going from FiniteDel to BoundDel as in verification tests. However, the choice between BoundDel and ExactDel does not seem to be a relevant factor for resulting runtime. For all three algorithms a similar effect of delay bound on synthesis runtime can be observed as for verification, in that the problem tends to become more difficult with higher bounds.

As opposed to verification, adding more observables in the case of synthesis has a clear effect on runtime for all synthesis algorithms. The symbolic method and the enumerative method computing all minimal solutions show steep increases in runtime for most use cases. The symbolic method scales a bit better on average, reflecting the results in Figure 4.12. The Enumerative-Optimal method exhibits a more modest increase in runtime adding more observables. Interestingly, it is in some cases very sensitive to changes in the search and solution space. It shows in general also



(a) incrementally enlarging set of observables



(b) min/mean/max runtimes for samples of each cardinality

Figure 4.13: Effect of number of observables on synthesis runtime (WBS model, enumerative algorithm with costs).

an increase in runtime with more observables, but when the additional observables allow new optimal solutions with lower cost, the convergence can improve significantly. This behavior can be seen for instance for the WBS model in Figure 4.13a. These samples are from the additional benchmarks described in the next paragraph; they are incremental in the sense that every sample has a strict superset of the observables of the predecessors.

To test how the absolute number of observables relates to runtime we took two models, CassiniInf and wbs and computed for each 15 series of incremental subsets of observables (435 use cases for CassiniInf and 480 for wbs in total). We thus obtained, for each use case, 15 samples of observable configurations for various cardinality layers. Costs are fixed upfront between 1 and 8 for each sensor. For each sample we ran the three algorithms to synthesize all configurations that are FiniteDel-diagnosable. The results, which are also included in the plots of Figure 4.12, show that the run-times of all three algorithms has little relation to the absolute number of observable. This can be seen for instance in Figure 4.13b, which shows the minimum, maximum, and average runtime for Enumerative-Optimal at each cardinality level. The average runtime steadily increases with more observables, but there are large differences between the minimum and maximum values at each cardinality. The run-times increase faster for the other two algorithms, but the situation is similar. This suggests that, rather than the absolute number of observables, it is important how the solution space is positioned in the search space.

Finally, we would like to remark the importance of the efficiency of the Symbolic algorithm. It shows that using generic parameter synthesis procedures for monotonic problems as opposed to specific algorithms for synthesis of observables does pay off in terms of performance. The mapping makes it also possible to directly exploit any future advancements in parameter synthesis, which is a more generic problem and thus receives broader attention from the research community. The ability to efficiently compute all (minimal) diagnosable solutions is also significant as it directly enables selection criteria that go beyond monotonic cost functions. For instance, solutions that are optimal w.r.t. a non-monotonic cost function can be extracted from the symbolic expression returned by parameter synthesis by using a pseudo-Boolean constraint solver or any other reasoning engine that is able to deal with costs natively.

4.6 Related Work

Since the seminal work of [Sampath et al., 1995], diagnosability has been the subject of a large body of works, both for verification and for synthesis. In the following we compare the proposed approach to these works in terms of framework, approaches to verification, and approaches to synthesis.

4.6.1 Frameworks for Diagnosability

The framework adopted in this chapter is highly expressive: it encompasses finite and infinite-state systems, temporally extended conditions to be diagnosed, various forms of delay, and temporally extended diagnosis contexts to exclude unrealistic scenarios from the analysis. It is therefore possible to embed most of the other frameworks for diagnosability, which are specialized to subsets of these features, within it.

As a first remark, notice that the works in [Sampath et al., 1995, Jiang et al., 2001, Yoo and Lafortune, 2002b, Jiang and Kumar, 2002, Cimatti et al., 2003, Jéron et al., 2006, Rintanen and Grastien, 2007, Biswas et al., 2010, Ye and Dague, 2010, Madalinski et al., 2010, Li et al., 2015, Boussif and Ghazel, 2015] rely on the hypothesis of finite-state plant. An exception are the works in [Tripakis, 2002, Altisen et al., 2006, Xu et al., 2010, Biswas et al., 2006, Daigle et al., 2009, Bayoudh and Travé-Massuyès, 2014, Bresolin and Capiluppi, 2013, Di Benedetto et al., 2011], which work with dense-time models, and in [Morvan and Pinchinat, 2009, Chédor et al., 2014, Cabasino et al., 2012], which explicitly deal with specific classes of infinite-state transition systems. Dense-time modeling frameworks are out-of-scope for the present chapter, where we focus on discrete-time only. However, it is worth noting that in some approaches a discrete abstraction is proposed to deal with hybrid automata. In [Bayoudh and Travé-Massuyès, 2014], for instance, a hybrid automaton is abstracted to a purely discrete model in a way that preserves certain diagnosability properties of the original automaton. On the resulting abstraction then any algorithm for finite-state systems can be used.

By adopting LTL contexts to restrict the set of traces being considered, we can in principle enforce also fairness constraints on these traces and thus investigate diagnosability under fairness assumptions on the transition system. The use of fairness constraints is relevant only for FiniteDel, where we do not fix an a-priori bound for diagnosis. However, as noted in Section 4.1.2, the critical pair method in the case of such constraints can be used only for falsification and not for verification.

Classical approach and trace diagnosability

Most of the works on diagnosability refer to the framework of [Sampath et al., 1995], for instance [Jiang et al., 2001, Yoo and Lafortune, 2002b, Rintanen and Grastien, 2007, Morvan and Pinchinat, 2009, Biswas et al., 2010, Ye and Dague, 2010, Madalinski et al., 2010, Cabasino et al., 2012, Chédor et al., 2014, Li et al., 2015]. In these works, diagnosability requires detection of a fault event within a finite number of steps; a counterexample is a pair of infinite traces where one contains the fault event, the other one does not, and both produce the same observations ad infinitum.

On one hand our frameworks differs in that it uses temporally extended fault models and has various notions of delay. Furthermore, the definition of diagnosability given in [Sampath et al., 1995] might be stronger than necessary, since it is defined as a global property of the plant. Imagine the situation where a few traces are not diagnosable because they represent a scenario that is possible but not realistic, or to be excluded by assumption. The presence of these traces breaks system diagnosability. By using the concepts of trace diagnosability we redefine diagnosability from a global property expressed on the plant to a local property on single traces. With a context specification we can then easily exclude these traces, as shown in Section 4.1, and better characterize the scenarios where we require diagnosability.

In [Ye et al., 2016] the notion of *manifestability* is introduced; a fault is manifestable if there is at least one trace on which it can be diagnosed. In some sense it takes trace-diagnosability to the extreme and tries to check whether diagnosis is ever possible.

Temporally extended diagnosis conditions

Extensions of the framework of [Sampath et al., 1995] that model faults as temporally extended properties instead of simple events are presented in [Jiang and Kumar, 2002, Jiang et al., 2003a, Jéron et al., 2006, Bozzano et al., 2014a]. In the present framework we adopt the notion of diagnosis condition of [Bozzano et al., 2014a].

The most basic extension is described in [Jiang et al., 2003a], which requires an alarm to be raised after an occurrence of k faults of the same fault class; the authors also describe specialized checking procedures. The framework of [Jéron et al., 2006] is more expressive and uses diagnosis conditions of the form $\mathsf{O}\beta$, where β is a safety property. The approach is less general than our fault models which use generic Past-LTL. It can for instance not verify whether a diagnoser will be able to generate the alarm for all occurrences of a temporary condition along any trace, just for fixed number of occurrences.

Finally, in [Jiang and Kumar, 2002] a fault is a generic LTL property. This framework is not comparable to ours. On one hand, full LTL is obviously more expressive than LTL with only past operators. However, the LTL fault formula is evaluated only at the first point of the trace and not at every point of it as in our case.

Delay bound requirements

Definition 15 is a generalization of Sampath's definition of diagnosability:

Theorem 18. *[Bozzano et al., 2015c] Let S be a plant such that there is no cycle of unobservable events, and let p be a propositional formula, then p is diagnosable (as defined in [Sampath et al., 1995]) in S iff there exists d such that* $\textsc{BoundDel}(\circ, \mathsf{O}p, d)$ *is diagnosable in S.*

This notion of delay is used in all related works on finite-state systems, where it is equivalent in asking whether the condition β is FiniteDel diagnosable. In fact, the counterexamples that are being looked for in the related work match our definition of FiniteDel critical pairs.

In the present chapter we explicitly distinguish between BoundDel and FiniteDel in order to study specialized proof methods, which becomes important when going beyond finite state spaces and system-diagnosability. Furthermore, by defining diagnosability on individual points of individual traces and allowing non-monotonic diagnosis conditions, the delay requirement must be evaluated in every one of those points instead of only the first one where the condition occurs. We also allow the specification of ExactDel alarm conditions, a new type of delay constraint introduced in [Bozzano et al., 2014a].

Dynamic and uncertain observations

Instead of operating with a static set of observables, in [Cassez et al., 2007] and [Wang et al., 2008] it is assumed that sensors can be enabled and disabled during execution of the plant to save operating costs. The goal is to minimize the number of active sensors at any time during the plant's run, and turn on only those that are strictly necessary to preserve diagnosability of the fault event. In the present chapter we only consider a static choice of observations (sensors are always switched on).

Diagnosability w.r.t. masks over observable events is a way to model uncertainty in observations and is discussed in various works [Jiang et al., 2003b, Cassez et al., 2007, Su et al., 2016]. In [Su et al., 2016] these masks are described as *logically uncertain observations*, i.e. observations that cannot be discriminated. The authors describe also the class of *temporally uncertain observations*, that is when the order in which observable events were generated is not clear.

In our framework the observations are regarded as certain, but to some extent these uncertainties can be modeled also here. To realize *logically uncertain observations* one could introduce new observable signals that are functions over the original observables plus possibly an uncertainty factor such as sensor faults or amount of signal noise. Also *temporally uncertain observations* can in principle be modeled in our framework by using buffers that non-deterministically delay the output signal. Furthermore, by using the synthesis procedures described in the present work we can also reason over these uncertain observable signals to find the most cost-efficient level of sensor precision that still guarantees diagnosability.

Distinguishing sets of states

The framework in [Cimatti et al., 2003] deals with finite-state systems, and expresses the diagnosis condition as a non-temporal pair of conditions to be separated with delay 0; in [Bittner et al., 2012] and [Boussif and Ghazel, 2015] it is extended with different notions of delay. This approach is incomparable with the choices made in the present chapter. On one hand, by using temporal specifications we can obtain richer fault models, and we also have several ways to express delay requirements; but then again we do not express diagnosability as being able to distinguish a *pair* of properties. The focus here is on the diagnoser that one will eventually build, and thus on the condition for which it will need to generate an alarm. The approach chosen in this chapter was found to be simpler and yet adequate in practical cases. The xSAP and COMPASS tool-sets for instance are based – as far as diagnosability is concerned – on the theoretical framework expressed in this chapter. A similar motivation applies also to use an LTL context referring to the traces of the original plant, as opposed to referring directly to twin-plant traces as done in [Cimatti et al., 2003]; indeed trace diagnosability refers to single traces, not pairs of traces.

4.6.2 Verification of Diagnosability

Verification of diagnosability has been studied in a variety of works. In [Sampath et al., 1995] the classical definition of diagnosability for DES and an algorithm to check it are given, based on building a diagnoser for the system under observation. The diagnoser contains cycles of ambiguous belief states if and only if the condition is not diagnosable.

Since the diagnoser method is exponential in the number of system states, in [Jiang et al., 2001] the twin-plant approach is introduced as a more efficient alternative that is polynomial in the number of states. The key idea is to avoid building the diagnoser and instead only compare pairs of indistinguishable infinite traces by exploring the twin-plant. The system is diagnosable iff no such pair exists. In [Yoo and Lafortune, 2002b] a similar method is described. This twin-plant based approach has become the standard way to study diagnosability. Variations of the method have been studied in [Jéron et al., 2006, Cabasino et al., 2009, Morvan and Pinchinat, 2009, Madalinski et al., 2010, Cabasino et al., 2012, Chédor et al., 2014].

In order to avoid using ad-hoc constructions as in the methods above, in [Jiang and Kumar, 2002] the problem of diagnosability is reduced to LTL model-checking. This strategy is also adopted in [Cimatti et al., 2003], which furthermore uses symbolic techniques to address the problem of state explosion; this is the general approach that we also adopt in the present chapter and validate empirically. Another symbolic approach to diagnosability to deal with state explosion, based on the twin-plant, is also advocated in [Rintanen and Grastien, 2007], which proposes a SAT encoding for falsification of diagnosability, similar to bounded model-checking.

In [Grastien, 2009] a specialized backward-directed algorithm is proposed starting from an ambiguous state and trying to reach an initial state. The intuition here is that when a fault occurs and it quickly produces observable symptoms that cause divergence of traces, a backward search should reach a fix-point much sooner than a corresponding forward search, thus saving computation time. The same idea is reformulated in [Li et al., 2015].

Compositional approaches that aim at dealing with the complexity of bigger systems are described in [Schumann and Pencolé, 2007, Ye and Dague, 2010], where the results of local twin-plant checks for individual components are integrated to infer the global diagnosability property. In comparison, we adopt a monolithic approach and try to address state-explosion via symbolic model-checking.

In this chapter we have shown that when increasing the expressiveness of the diagnosability framework, the standard twin-plant method cannot be used in all cases as a verification method (though it always works for falsification). This opens interesting opportunities for future research.

Diagnosability via Temporal-Epistemic Logic

Epistemic logic has been used to describe and reason about knowledge of agents and processes. There are several ways of extending epistemic logic with temporal operators, and in [Bozzano et al., 2014a] the logic KL_1 [Halpern and Vardi, 1989] is used to reason about an ideal diagnoser. KL_1 extends LTL with the epistemic operator K. The intuitive semantics of $\mathsf{K}\beta$ is that the diagnoser *knows*, by only looking at the observable behavior of the system, that β holds in the current execution. The definition of the semantics of K can take into account multiple aspects such as observability, synchronicity and memory. A semantics for ASL based on Temporal Epistemic Logic (TEL) has been described in [Bozzano et al., 2015c], providing a sound and complete technique for performing diagnosability testing using Temporal Epistemic logic model-checking. For example, the diagnosability test for $\textsc{ExactDel}(A, \beta, n)$ consists of the KL_1 formula $G(\beta \rightarrow X^n K Y^n \beta)$, stating that whenever β occurs, exactly n steps afterwards, the diagnoser *knows* that n steps before β occurred. Since K is defined on observationally equivalent traces, the only way to falsify the formula would be to have a trace in which β occurs, and another one (observationally equivalent at least for the next n steps) in which β did not occur; but this is in contradiction with the definition of diagnosability as given in Definition 14.

In many practical situations for which the critical pairs approach is complete for diagnosability verification, the twin-plant construction presents several advantages over the TEL encoding. First, the level of maturity of tools and techniques for LTL model-checking is significantly higher than for TEL. This is particularly true for what concerns SAT/SMT based algorithms for infinite-state systems that TEL model-checkers are mostly lacking, and when available support only some particular semantics [Cimatti et al., 2016], or solve a problem that is as hard as building the diagnoser for the system, thus defeating the purpose of performing the diagnosability test as a validation step. Second, encoding the concept of context in the specification is much more difficult in the TEL framework where the context must be embedded within the plant model, in order to provide a different interpretation of the K operator. Third, the twin-plant framework provides a clean way to go from verification to synthesis. On the contrary, in the TEL framework, the synthesis problem would need to be encoded as a problem of finding an agent that satisfies the property. That, however, is a problem that has never been studied.

4.6.3 Synthesis for Diagnosability

Also synthesis for diagnosability has been studied in various works, mostly for finite-state systems, and usually with non-symbolic (explicit-state) approaches.

In [Debouk et al., 2002] an optimal way to navigate the search space of sensor configurations is presented in search for a cost-optimal solution. A-priori probabilities for a configuration to be diagnosable are used as a bias to guide search. The approach is independent of the specific system dynamics at hand, as it takes the diagnosability test as a black-box, but has restrictive assumptions on the cost of configurations: any configuration with cardinality k has lower cost than any configuration with cardinality k+1, and within each cardinality layer, a total order in terms of costs exist. In the present work we do not assume such restrictions, and the cost of a configuration is simply the sum of the cost of each sensor.

In [Jiang et al., 2003b] two general algorithms are described for computing optimal sensor sets for different observation problems. For diagnosability, the authors rely on the verification approach of [Jiang et al., 2001]. The first strategy is top-down in that it checks for each sensor whether it can be removed without breaking diagnosability. If so, the sensor is removed. The second strategy is bottom-up; it adds sensors until the system becomes diagnosable, then applies the first method. The result of both procedures is one minimal sensor configuration.

In [Briones et al., 2008] the two algorithms of [Jiang et al., 2003b] are improved by directly exploiting critical pairs to discover observability requirements. The idea is the same as in Equation 4.1, except that in our case a more generic definition is needed to account for all possible alarm specifications. It is not described how to integrate this technique with an efficient verification engine. In [Santoro et al., 2014] this idea of bottom-up search with analysis of critical pairs is extended to identify all minimal solutions. It does however not apply the minimization step (top-down) and thus might check more configurations than necessary. The algorithm iterates until all solutions have been enumerated.

Building on the idea of [Briones et al., 2008] of extracting observability requirements from critical pairs, [Grastien, 2009] proposes to use best-first search to identify a cost-optimal solution. The idea is to check the best candidate at every turn. If it makes the system diagnosable, a cost-optimal solution has been found; else, the observability requirements are extracted from the critical pair and all candidates that do not satisfy them are removed from the search space. LTL model-checking on the twin-plant is proposed as a verification back-end, similar to [Cimatti et al., 2003]; implementation strategies for managing the search space are not discussed.

Our own previous work [Bittner et al., 2012] on synthesis for diagnosability follows the framework of [Cimatti et al., 2003]. We introduced the notion of parameterized twin-plant as a way to reason about diagnosability under different observables using a single model; this is a central feature of our approach, as it enables incrementality features in solvers across various calls. Two algorithms are described. The first one corresponds to algorithm Enumerative-Optimal of Section 4.5, except that we use different proof obligations. The second one computes all diagnosable configurations and minimizes afterwards; it is similar to the parameter-synthesis approach proposed here, but does not exploit the monotonicity of the problem as efficiently (see [Bozzano et al., 2015d]).

In [Bittner et al., 2014a] a symbolic approach for synthesis of Pareto-optimal parameter assignments is presented, with synthesis of observables for diagnosability as defined in [Bittner et al., 2012] as an application example. It explores the lattice induced by cost functions top-down. The efficiency of this approach relies on the reduction of diagnosability to invariant checking and the use of inductive invariants for fast reduction of diagnosable sensor configurations. The approach is thus not directly applicable to our LTL formulation of the problem.

4.7 Summary

In this chapter we proposed a comprehensive approach to the problems of verification of diagnosability and synthesis for diagnosability. We formally defined an extended version of the problems, starting from expressive patterns that take into account various forms of delay in the diagnosability and the operating conditions. The adequacy of the critical-pair method to verify diagnosability as the standard approach in the literature is studied in detail. For many important fragments of the framework, the absence of critical pairs is indeed a necessary and sufficient condition for diagnosability, but in some cases it is only a necessary one.

As opposed to most related works, we rely in all cases on off-the-shelf symbolic LTL model-checking as a way to identify critical pairs, prove the absence thereof, and synthesize optimal sensor configurations, instead of implementing specialized procedures. This enables our approach to automatically benefit from any advancement made in this field, for both finite-state and infinite-state transition systems.

We provide efficient implementations of two enumerative best-first algorithms that analyze critical pairs to discover observability requirements; the first one uses sensor cost as optimization criteria, the other one set-inclusion. These two algorithms are compared to a symbolic off-the-shelf implementation of parameter synthesis with support for monotonic problems.

With a comprehensive experimental evaluation based on realistic models we also demonstrated the practical applicability of the proposed algorithms to a variety of problems formulated with the new framework, for verification and for synthesis. The performance of the proposed symbolic approach to compute all diagnosable sensor configurations shows the feasibility of optimization w.r.t. criteria going beyond (monotonic) cost functions.

There are many promising directions for future work. First we will investigate a possible extension of the critical-pair method to falsify diagnosability, in order to cover also the fragments where the absence of such pairs is only a necessary condition, possibly extending them to a notion of "critical sets". This will also include studying corresponding implementations, possibly based on LTL model-checking as opposed to epistemic model-checking.

Second, we will extend the framework by relaxing some of the assumptions, to include a form of asynchronous composition between the diagnoser, in the style of [Bozzano et al., 2015c], to allow for an asynchronous integration in the overall system architecture. A very important concept is also the one of bounded recall, of particular concern in systems with very limited computational resources.

Third, we will consider lifting the proposed approach from the case of transition systems to the case of hybrid systems, that integrate continuous dynamics [Henzinger, 2000] and a logic such as HRELTL to express requirements over continuous traces [Cimatti et al., 2015b]. We will adopt some recent SMT-based techniques, that are becoming relevant for the analysis of networks of hybrid automata [Cimatti et al., 2012, Mover et al., 2013, Cimatti et al., 2015a]. With this step we will also harmonize the time model with the TFPG framework which we also plan to update to the hybrid setting; this will allow us to work on the same models with the same interpretation of time.

Finally, we plan to investigate other forms of synthesis for diagnosability, for example to find the minimum delay in an alarm condition. In many practical situations, we are interested in knowing the minimum value of d for which we have e.g. BoundDel diagnosability; this induces an associated optimization problem. A more complex problem is to synthesize the individual sensors (e.g. choosing against which thresholds variables should be monitored), driven for example by considerations based on the sensor fault tolerance or noise tolerance.

Chapter 5

Industrial Application

In this chapter we describe the application of the techniques presented in the previous chapters in two different projects, in which we were able to study their adequacy in an industrial setting. Before describing the detailed contributions in the following sections, we give a brief overview of these projects.

FAME

In the FAME project of the European Space Agency – FAME is short for "Failure and Anomaly Management Engineering" – a novel, model-based, integrated process for FDIR design was proposed [European Space Agency, 2011, FAME, 2016]. It aims at enabling a consistent and timely FDIR conception, development, verification and validation. The process is supported by a model-based toolset covering a wide range of formal analyses. In [Bittner et al., 2014b] the project and its results are described in more detail. Here we give a brief overview to contextualize our contributions.

The problem FAME tried to address is that often the quality of FDIR implementations is suboptimal due to a lack of standardized development guidelines (development phases covering the whole life-cycle, key documents as inputs and outputs of these phases). Furthermore it becomes

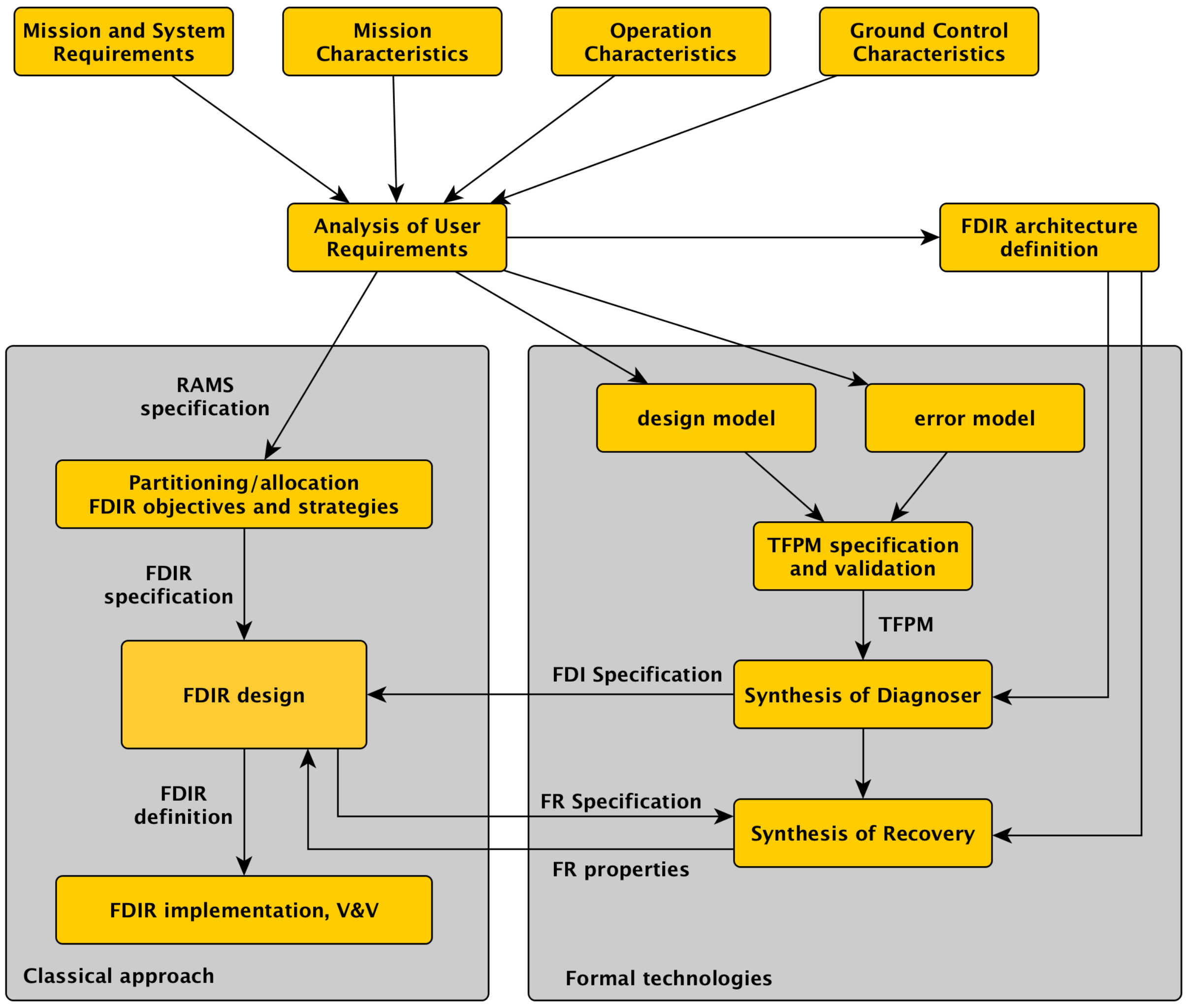


Figure 5.1: FAME Process Overview

more and more difficult to deal with the increasing complexity using traditional informal analysis means, especially considering the amount of potential failure scenarios. FAME thus proposed a clearly structured FDIR development process, with key documents and artefacts, integrated with formal model-based analysis technology.

Figure 5.1 shows the overall FAME process. The main steps are the following.

Analyze User Requirements All user requirements that impact the design of the FDIR are collected, including RAMS (reliability, availability, maintainability, and safety) and autonomy requirements. Also a mission phase and spacecraft operational mode matrix is built, as requirements might apply only to specific combinations of those. For instance, autonomy requirements might not be the same for each mission phase.

Define Partitioning/Allocation RAMS and autonomy requirements are allocated per mission phase / operational mode. Moreover, the FDIR architecture is modeled, including identification of functional decomposition, sub-system HW/SW partitioning, sub-system functions and redundancy, integration of pre-existing FDIR functionalities, and definition of FDIR levels. Also the distribution of the FDIR functionalities is taken into account: FDIR management can be decentralized, hierarchical or a combination of both.

Perform Timed Failure Propagation Analysis In this step a timed failure propagation model (TFPM) is developed, based on inputs from preceding RAMS analyses such as FTA, FMEA, hazard analysis, and observable fault symptoms. Also diagnosability properties of the TFPM are analyzed.

Define FDIR Objectives and Strategies Starting from RAMS and autonomy requirements and exploiting the previous results on FDIR and failure propagation analysis, in this step the system-level FDIR objectives (such as required behavior in presence of failures) and subsystem-level FDIR strategies (steps to be performed given fault detection and respective objectives) are established.

Design In the design phase the detailed FDIR implementation is defined (identification of parameters to be monitored, ranges, isolation and reconfiguration actions), along with a detailed software specification (suitability of standard message passing services, definition of additional services) and spacecraft database (SDB) specification (insertion of monitoring information, definition of recovery actions, link between monitoring and recovery actions).

Implement FDIR, V&V The last step is concerned with the implementation of FDIR in hardware and/or software, and its verification and validation with respect to the specifications.

Note how on the process level the term "timed failure propagation *model*" is used, thus separating process from supporting technology, in our case TFPGs. From the process point-of-view it is thus not required that TFPGs are the propagation models to be used, but it is important to have a system-wide model of timed failure propagation that serves as a central reference of the *failure behavior* the FDIR needs to deal with. As opposed to adopting the more static notion of *failure modes*, which can be interpreted as system states, propagation modeling focuses on the sequences of failure effects throughout the system.

Research Stay at ESA/ESTEC

Further contributions on the topics of this dissertation were produced during a 10-month research stay at ESA ESTEC in Noordwijk (The Netherlands), in which we studied the application of TFPGs as a failure analysis in "Solar Orbiter" (SOLO).

Solar Orbiter is a Sun-observing satellite under development by the European Space Agency. It will orbit the Sun to perform various scientific observations which are very difficult or impossible to do from Earth. During its operation the satellite will come closer to the Sun than even the planet Mercury. Currently the launch is planned for October 2018.

The FDIR requirements on SOLO are much more stringent than on typical Earth-observing satellites, especially due to the intensity of solar radiation. One of the main challenges is to continuously protect the spacecraft by keeping a heat-shield oriented towards the Sun. Many faults are highly time-critical as they can quickly cause considerable damage to the spacecraft. Detection, isolation, and recovery need thus to be performed in extremely short time frames.

Three case studies (presented in the following sections) on timed failure propagation analysis were performed in collaboration with an ESA software engineer from the SOLO project. For this we analysed the project documentation related to FDIR (mostly FMECA and FDIR design coverage documents) and the general software/hardware architecture (for instance the Control Algorithm Specification). These documents were at a pre-CDR (Critical Design Review) level.

Collecting the necessary information for the case studies was very challenging especially due to the huge amount of documentation, which was several hundreds of pages just for the documents effectively used. Additionally we found that the information needed for the case studies was scattered throughout the documents, and collecting and interpreting everything required substantial work and several interactions with engineers. A formal and structured approach to interpret this information, for instance by using TFPG modeling, is thus a clear benefit and increases the confidence in the completeness of the analysis.

Based on the experience made in these case studies we were thus able to evaluate the application of TFPGs in a real project. The proprietary information that we used for our study is subject to non-disclosure, and therefore cannot be quoted literally in this thesis. However, we remark that the models created for the case studies are generic; the problems the case studies deal with are quite universal and do not apply only to SOLO.

Chapter Contributions In this chapter we document the following contributions.

- In Section 5.1 we describe a method to translate a TFPG into a transition system specified in SMV, such that it can be used as a basis for diagnoser synthesis. This was developed for FAME.
- In Section 5.2 we present two TFPG case studies for SOLO: one looking at error propagation within software, and one studying failure propagation during detection and isolation activities. The results demonstrate the adequacy of the developed framework and implementation.
- In Section 5.3 we investigate the suitability of TFPGs as a way for system-wide propagation modeling as proposed in FAME. We develop a use case for one particular propagation chain in SOLO. Furthermore, based on the case study, we make some observations of using TFPGs as a way to assess and tune FDIR design coverage and to support diagnostic activities during testing and operations.

In Sections 5.2 and 5.3, which cover case studies on SOLO, we also describe how the analyses helped to raise five issues that were submitted to the FDIR critical design review. Four of them were classified as major, one as minor. The issues led in most cases to improvement of the available design documentation and in one case modification of the design, as the analysis unveiled a missing consistency check.

We summarize our findings in Section 5.4 and give an outlook on possible future work in terms of further case studies and practical applications.

5.1 Enabling Diagnoser Synthesis via TFPGs

In FAME, TFPGs are used to capture the timed failure propagation behavior of a system model that describes both nominal and faulty dynamics. Not only are they used as a failure analysis like FTA and FMEA, they also serve directly as an input to the synthesis of a diagnoser, thus bridging the gap between failure analysis and FDIR implementation with formal model-based techniques. This positioning of TFPGs in the FAME flow can be seen in Figure 5.1. The TFPG is produced and validated (or generated automatically) using the algorithms of Chapter 3, and then used to automatically build a diagnoser. The contribution described in this section is a translation method from TFPG to transition systems expressed in the SMV language, which enables this automated step from TFPGs to the synthesis of a diagnoser.

The diagnoser synthesis approach used in FAME is based on the framework of [Bozzano et al., 2015c]. It is similar to the framework used in Chapter 4, with the exception that, in order to accommodate the asynchronous modeling style used in FAME, the composition between diagnoser and system model is asynchronous. Time is assumed to advance during tick events, and no time is assumed to pass during other transitions. The diagnoser is synchronized only on observable transitions including ticks: upon receiving these interrupt signals, its state is updated and alarm signals are raised accordingly.

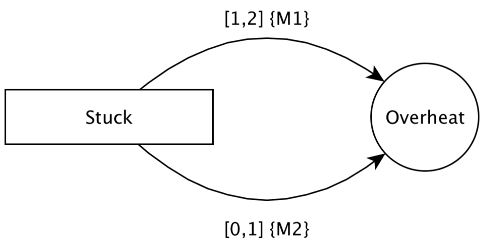
As a TFPG is effectively an abstraction of a more detailed system model and thus a simpler model, the diagnoser synthesis task is easier. Furthermore the resulting diagnoser will be smaller, as it doesn't have to reason over all state variables. Being an abstraction though can at the same time be a disadvantage, as less information is available to the diagnoser, which thus might produce less accurate diagnoses.

For diagnosis via TFPGs also the approach in [Abdelwahed et al., 2009] can be used. As new observable discrepancies activate or timeouts trigger on expected discrepancy activations, the reasoning algorithm is run to update the set of hypothetical TFPG states. The difference w.r.t. the diagnosis approach in FAME is that the diagnostic reasoning is pre-compiled into the diagnoser data structure. It contains rules to update its state (transition rules) based on new observations that can be applied in constant time. This on the other hand directly allows not only the diagnoser itself to be formally verified, but also the integrated product of diagnoser and system model. The latter is not straightforward with diagnosis directly reasoning over the TFPG, as algorithm and model are separate and would have to be translated to a unified formal model on which verification can be performed. The synthesized diagnoser however *is* the formal model, and thus the gap between verification model and implementation is much smaller. Furthermore, with constant-time updates it should also be easier to estimate worst-case execution time for the synthesized diagnoser. On the other hand the update time (and synthesis time) might be unreasonably large for bigger models, in addition to a more expensive synthesis step.

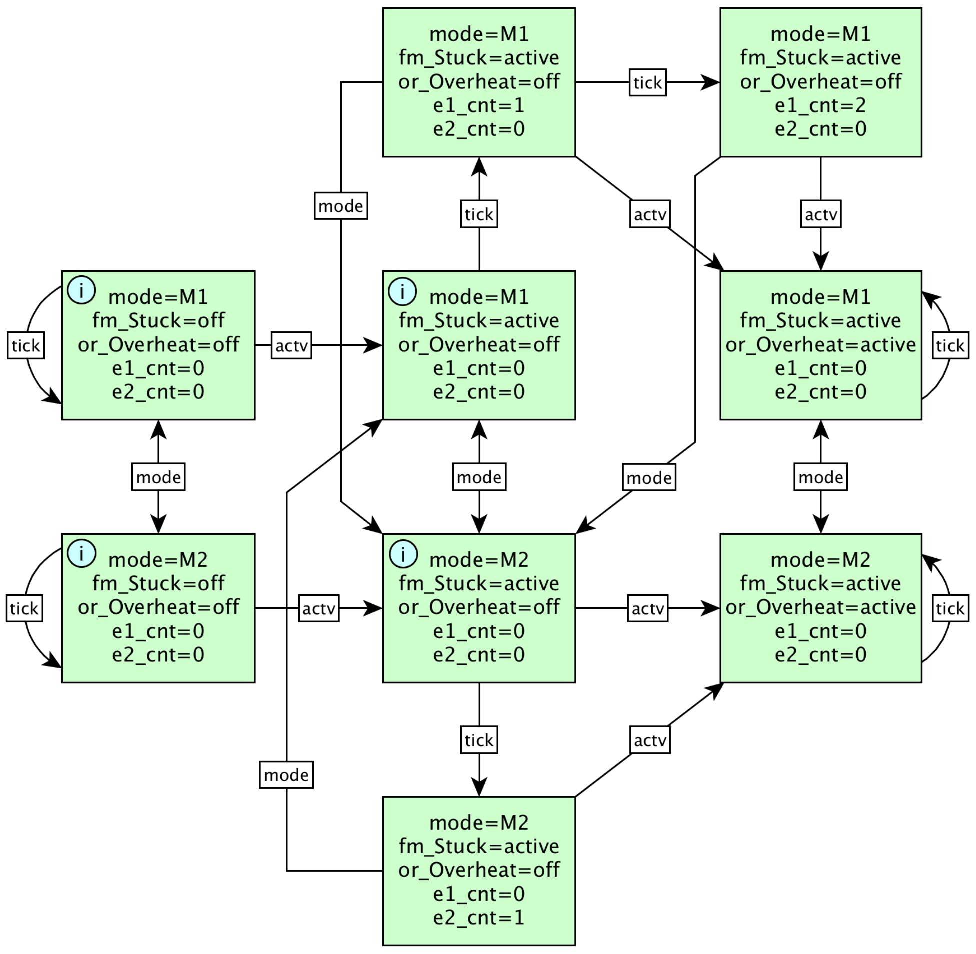
5.1.1 TFPG-to-SMV Translation

We describe now how a TFPG can be translated into an SMV model. We use the TFPG shown in Figure 5.2a as a running example for this section to illustrate individual steps. The state-space of the SMV code that we will generate is shown in Figure 5.2b, and the complete code itself can be found in Appendix A.

The state variables of the translation follow the definition of TFPGs (Definition 1) and comprise the system mode, edge timers counting for how long an edge has been active, and the activation status of failure



(a) sample TFPG


mode=M1 fm_Stuck=active or_Overheat=off e1_cnt=1 e2_cnt=0
mode=M1 fm_Stuck=active or_Overheat=off e1_cnt=2 e2_cnt=0
i mode=M1 fm_Stuck=off or_Overheat=off e1_cnt=0 e2_cnt=0
i mode=M1 fm_Stuck=active or_Overheat=off e1_cnt=0 e2_cnt=0
mode=M1 fm_Stuck=active or_Overheat=active e1_cnt=0 e2_cnt=0
i mode=M2 fm_Stuck=off or_Overheat=off e1_cnt=0 e2_cnt=0
i mode=M2 fm_Stuck=active or_Overheat=off e1_cnt=0 e2_cnt=0
mode=M2 fm_Stuck=active or_Overheat=active e1_cnt=0 e2_cnt=0
mode=M2 fm_Stuck=active or_Overheat=off e1_cnt=0 e2_cnt=1
tick
actv
mode

(b) state-space of SMV translation

Figure 5.2: Sample TFPG with state-space of corresponding SMV translation. $e1$ is the upper edge with the guard $[1,2]\{M1\}$, $e1$ the lower edge with the guard $[0,1]\{M2\}$. Possible initial states are marked with a circled i.

modes and discrepancies corresponding to the graph nodes. Furthermore we assign one label to each transition with an input variable. On top of these variables we then define a number of transition rules in SMV as follows.

Transition Types

There are three types of transitions: node-activation, mode-change, and time-tick. All changes made during a discrete transition of the SMV model must belong to one of these transition types, and no two changes during one transition can belong to different transition types. This guarantees the assumption that discrete changes in the corresponding TFPG state occur only during untimed transitions. The following SMV code represents this label:

```
  IVAR trans_type : {NODE_ACTIVATION, MODE_CHANGE, TIME_TICK};
```

During a *node activation* transition, all node activations are performed that are mandatory, and non-deterministically also any one that is possible. A *mode change* transition updates the system mode. When only one mode exists, mode change transitions never occur. If only one system mode is mentioned in the TFPG, the mode change label can be dropped. Finally, a *time tick* represents the passing of one discrete time unit; with respect to time, node activations and mode changes are regarded as instantaneous, i.e. they don't happen during a timed transition.

For the translation we discretize the time intervals in the TFPG, dividing each constant by a fixed sampling interval δ. For the example we use $\delta := 1$. The system state is thus sampled, via TFPG abstraction, every time unit; the passing of this time unit is represented by the tick transition.

System Mode

The system mode is declared as an enumerative type in the main SMV module.

```
  VAR system_mode : {MODE1, MODE2, ...};
```

No constraints for mode switches are specified, i.e. mode switches can change the current mode to any other mode. However, to enforce that during a mode-change transition a new mode is actually selected (in cases where multiple possible modes exist), we add the following transition constraint to the main SMV module.

```
TRANS trans_type = MODE_CHANGE <-> system_mode != next(system_mode);
```

next is an SMV keyword that causes an expression to be evaluated in the state after a transition. The expression `system_mode != next(system_mode)` thus states that the system mode in the current state is different from the system mode in the next state.

Failure Mode Nodes

Failure mode nodes are represented using the following SMV modules:

```
MODULE failuremode (trans_type)
  VAR status : {OFF, ACTIVE};
  ASSIGN init(status) := {OFF, ACTIVE};
  ASSIGN next(status) := case
    trans_type != NODE_ACTIVATION : status;
    status = OFF : {OFF, ACTIVE};
    TRUE : status;
    esac;
```

The status of a `failuremode` can change to active, during transitions of type node activation, independently from any other state variable. Once a failure mode – or also a discrepancy node, as shown below – is activated, it remains activated forever.

For the running example we declare the following `failuremode` node.

```
VAR failuremode_Stuck : failuremode (trans_type);
```

Discrepancy Nodes

Discrepancy nodes are represented in a similar way, but have additional activation constraints depending on the edges that connect to them. The SMV module of discrepancies is specific to the node semantics and the number of incoming edges. The following code shows the declaration of an `OR` node with two incoming edges, using a naming convention according to node type:

```
MODULE or_node_2 (edge1, edge2, trans_type)
```

For an `AND` node with three incoming edges the module header is:

```
MODULE and_node_3 (edge1, edge2, edge3, trans_type)
```

For the small running example we will declare the following node:

```
VAR or_node_Overheat :
      or_node_2 (edge_Stuck_to_Overheat_1, edge_Stuck_to_Overheat_2, trans_type);
```

Next we define for each type of module a `can_fire` and a `must_fire` signal, which tell whether the node *can* or *must* be activated, depending on the status of connected edges. For the `OR` node with two incoming edges, as in the running example, they will be defined as follows:

```
DEFINE can_fire := (edge1.can_fire | edge2.can_fire) & !must_fire;
DEFINE must_fire := edge1.must_fire | edge2.must_fire;
```

In words, an `OR` node *can* fire if at least one incoming edge can fire, but none must fire. An `OR` node *must* fire if any incoming edge must fire. Note that the two signals are modeled to be mutually exclusive for obtaining the correct behavior of the activation status variable as defined below. This is guaranteed by adding, for `OR` nodes, a corresponding explicit constraint.

For `AND` nodes these signals are defined as follows, assuming three incoming edges in this example. The two signals are mutually exclusive without additional constraints.

```
DEFINE can_fire :=
    (edge1.can_fire | edge1.must_fire) &
    (edge2.can_fire | edge2.must_fire) &
    (edge3.can_fire | edge3.must_fire) &
    (edge1.can_fire | edge2.can_fire | edge1.can_fire);
DEFINE must_fire := edge1.must_fire | edge2.must_fire | edge1.must_fire;
```

In words, an `AND` node *can* fire if at least one incoming edge can fire and all others can or must fire. An `AND` node *must* fire if all incoming edges must fire.

The rest of the SMV code of discrepancies is common for both `OR` and `AND` nodes, and implements the activation status.

```
VAR status : {OFF, ACTIVE};
ASSIGN init(status) := case
    trans_type != NODE_ACTIVATION : status;
    can_fire : {OFF, ACTIVE};
    must_fire : ACTIVE;
    TRUE : status;
    esac;
```

For an `OR` node d, a propagation can reach or activate d iff some edge $e = (v, d)$ has been active for at least $tmin(e)$ (`can_fire` signal) and at most $tmax(e)$ (`must_fire` signal) discrete time units. This guarantees that the $\psi_{\texttt{OR}\cdot A}$ proof obligation holds for d. When an edge $e = (v, d)$ is active for exactly $tmax(e)$ time units, then time will not pass before either d activates, or the system switches to a mode where the edge is not active. This guarantees that the $\psi_{\texttt{OR}\cdot B}$ proof obligation holds. For `AND` nodes the situation is similar, except that node activation can occur if all incoming edges can or must fire, and node activation must occur if all incoming edges must fire.

To guarantee that during node-activation transitions some node (failure mode or discrepancy) actually gets activated, we add the following transition constraint, similar to the one for mode changes. In our example we instantiate this rule for the nodes `failuremode_A` and `or_node_B`.

```
TRANS trans_type = NODE_ACTIVATION <->
  failuremode_{ID0}.status != next(failuremode_{ID0}.status) |
  or_node_{ID1}.status != next(or_node_{ID1}.status) |
  and_node_{ID2}.status != next(and_node_{ID2}.status) |
  ...;
```

To disable timed transition when some node *must* fire, we use the following constraint. In the example this involves only the node `or_node_B`.

```
TRANS (or_node_{ID1}.must_fire | and_node_{ID2}.must_fire | ...)
      -> trans_type != TIME_TICK;
```

Edges

Edges are used to identify whether a propagation from the source node to the target node can occur, and effectively implement the timed behavior. Their module signature is defined as follows.

```
MODULE edge (tmin, tmax, tmax_is_infinity, source, target, trans_type,
             system_mode_is_compatible)
```

An edge is declared by setting the `tmin` and `tmax` parameters to the respective constants, `source` and `target` to the respective node variables, `trans_type` to the respective variable from the main SMV module, and `system_mode_is_compatible` to a Boolean expression that indicates if the edge is active in the current system mode. If there is only one possible system mode or if the edge is compatible with all system modes, this last parameter can be set to the constant `TRUE`.

For cases where $tmax = \infty$, the parameter `tmax_is_infinity` is set to true; this allows the firing to be delayed ad-infinitum. `tmax` is set to the same value as `tmin`, to limit the range of the counter variable.

Two edges are declared in the example, as follows.

```
VAR edge_Stuck_to_Overheat_1 :
      edge (1, 2, FALSE, failuremode_Stuck, or_node_Overheat,
            trans_type, system_mode=M1);
VAR edge_Stuck_to_Overheat_2 :
      edge (0, 1, FALSE, failuremode_Stuck, or_node_Overheat,
            trans_type, system_mode=M2);
```

An edge is active if its source node has been activated by some preceding propagation, the system is in a compatible mode, and the target node hasn't been activated yet by a parallel propagation. This implies that once a discrepancy node is marked as activated, all incoming edges are marked as inactive because the propagation across them cannot occur anymore.

```
DEFINE is_active := source.status = ACTIVE &
                    system_mode_is_compatible &
                    target.status = OFF;
```

Furthermore, each edge module contains a counter variable that is used to measure the time elapsed from the last activation of the edge. The timer is disabled (set to 0) whenever the edge is not active.

```
VAR counter : 0..tmax;
ASSIGN init(counter) := 0;
ASSIGN next(counter) := case
    -- edge is not active, counter is reset
    next(!is_active) : 0;
    -- increment by one delta unit
    counter < tmax & trans_type=TIME_TICK : counter + 1;
    -- keep as-is
    TRUE : counter;
    esac;
```

Based on these counters we can now define when the edge can fire, i.e. when the propagation can occur.

```
DEFINE can_fire :=
    is_active & counter >= tmin & (counter < tmax | tmax_is_infinity);
```

```
DEFINE must_fire :=
    is_active & counter = tmax & !tmax_is_infinity;
```

In the special cases where $0 = tmin = tmax$ or where $0 = tmin$ and $\infty = tmax$ we don't need any counters and can use the following simplified SMV module definition.

```
MODULE instant_edge (source, target, tmax_is_infinity, system_mode_is_compatible)

  DEFINE is_active := source.status = ACTIVE &
                      target.status = OFF &
                      system_mode_is_compatible;
  DEFINE can_fire := is_active & tmax_is_infinity;
  DEFINE must_fire := is_active & !tmax_is_infinity;
```

The discrete edge timers on one hand enable the diagnoser synthesis technique of [Bozzano et al., 2015c], which uses them to reason on the time passing between observable discrepancy activations. On the other hand they bit-blast the range from 0 to `tmax`. If the chosen sampling interval δ is too small or the interval too big, the state-space of the resulting diagnoser will also be significantly bigger; in fact, diagnoser synthesis is exponential in the number of system states. But choosing a too coarse δ can have a detrimental effect on diagnosability, as the diagnoser will consequently work with less precise time measurements. A possible solution could be to drop timers where they don't affect diagnosability, i.e. where measuring the delay between observable events is not necessary to diagnose the condition of interest. For the purpose of diagnosis we would thus over-approximate the original TFPG. This will be investigated in future work.

TFPG Effectiveness Within the FAME project we also investigated applying the techniques of Chapter 4 to the asynchronous case used in [Bozzano et al., 2015c]. We focused on FiniteDel diagnosability of propositional

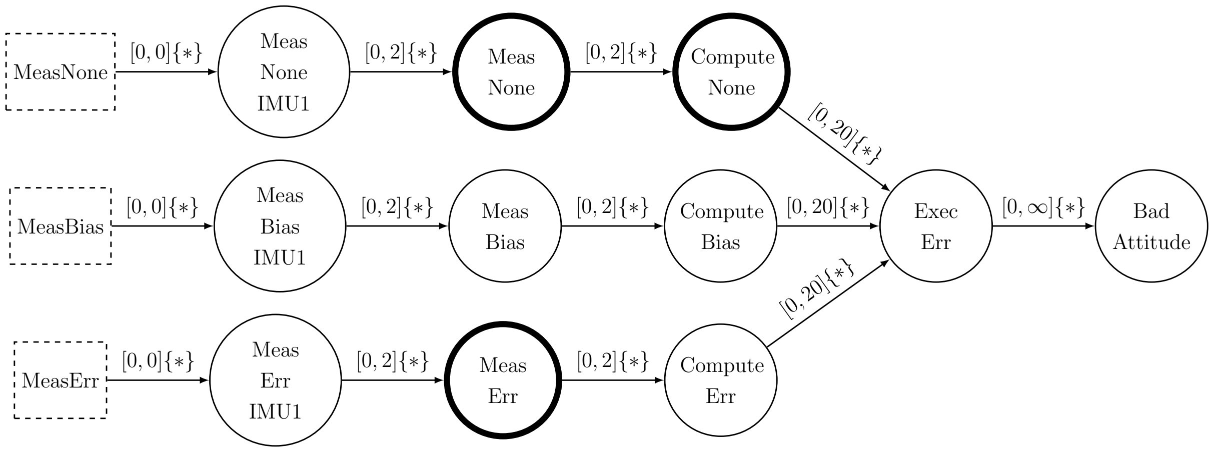


Figure 5.3: TFPG for the primary IMU in TGO. Bold circled nodes are monitored discrepancies used by the diagnoser.

diagnosis conditions, which effectively corresponds to the classical definition of [Sampath et al., 1995]. We used this to investigate diagnosability of the SMV translations of TFPGs as they were used for diagnoser synthesis. The twin plant was synchronized only on tick transitions and on observable discrepancy activations. Modes were assumed to be frozen, as no mode transition information is contained in a TFPG. This is an assumption often also made in practice, i.e. failure effects are analyzed assuming a stable system mode.

ExoMars Case Study: IMU Measurements We now briefly describe the case study done by Thales Alenia Space in FAME, documented in more detail in [Bittner et al., 2014b]. We focus on the results of diagnoser synthesis via TFPGs discretized using the SMV translation scheme described above. The case study involved the ExoMars Trace Gas Orbiter (TGO) mission, a spacecraft that was launched to study the atmosphere of Mars and to function as a relay station for a later mission. Specifically, the FAME analysis focused on the effect of IMU (Inertial Measurement Unit) faults on attitude control during Mars orbit insertion.

The TFPG shown in Figure 5.3 was developed to reason about several potential IMU-related failure events that could lead to a bad spacecraft attitude. It shows the propagation behavior in the primary IMU. The overall model contains also the backup IMU, for which a separate TFPG was developed (not shown here). A diagnoser was synthesized for a total of six alarm specifications, covering both IMUs. The state-space of this diagnoser had 2413 reachable states. Integrated with the system model of the orbiter, it was then used to proceed with design of the recovery strategies. The overall FDIR design was further analyzed using model-checking, and the effectiveness of the solution was verified. In particular it was shown that the FDIR component, when integrated with the system model, detected the faults according to the alarm specifications and led the system to a specific target configuration.

5.2 Focused Propagation Modeling

We now describe two case studies, done in the context of Solar Orbiter as a shadow engineering activity, in which we applied the TFPG algorithms of Chapter 3. Here we don't consider TFPGs in the context of the FAME process as in the previous section. Rather, the goal here was to study their general adequacy, in terms of usefulness to analyze propagation problems found in a real spacecraft design, in terms of performance, and in terms of the characteristics of the theoretical framework. Some of the feedback was also used to improve the TFPG synthesis algorithm.

As a basis for modeling we studied several FDIR-related documents of the SOLO project, such as FMECAs at unit, subsystem and system levels, FDIR design coverage documentation, and software design documents to understand the overall control architecture and logic. We studied propagation problems involving the Attitude and Orbit Control System (AOCS), which is responsible for realising the system-level requirement to keep the heat-shield facing the Sun when in close proximity. Hence the so-called "feared event" is a severe off-pointing during this mission phase which could lead to loss of the spacecraft. The AOCS uses, in a redundant configuration, sensors such as inertial measurement units (IMU), sun sensors (FSS), and star trackers (STR) to estimate the spacecraft's attitude and motion, and computes control commands to be sent to actuators such as the propulsion system (CPS) and reaction wheels (RWS). The analysis scope in this section is focused on specific components and scenarios, whereas in Section 5.3 we will focus more on the system perspective. With this we evaluate the TFPG analysis framework and implementation in different types of application scenarios.

The models were developed in SLIM, the modeling language of the COMPASS family of tools, to which also the FAME tool belongs. Subsequently they were translated to SMV, the language of nuXmv and xSAP which are the verification engines used by the COMPASS tools.

5.2.1 Case Study: Gyroscope Processing

The first case study involved modeling and analysis of the gyroscope channel processing function, a piece of software which reads different types of raw gyroscope sensor data coming from the IMU, retrieved over the bus and stored in the datapool. From these values the function computes the rotation rate around the axis on which the channel's gyroscope is positioned, along with health flags indicating whether data is corrupted, and how. In total the software function has 7 input variables and 13 output variables.

The function runs on the main computer and is called cyclically at fixed intervals. The overall function is composed of smaller subfunctions, some of which have internal state variables (in total 8) to store values computed in previous cycles for various purposes. Time elapses by one unit during tick events. After each tick, the values in the datapool are updated and the function uses them to compute, through various consecutive steps, the new output values. For the analysis purpose the duration of this computation is assumed to be instantaneous.

For this software function we created a model representing various computational steps on an abstract level. Variables with values in the reals are abstracted to discrete domains, such as “normal”, “degraded”, and “erroneous”. For degraded data readings we assume that the internal checks might or might not detect the corruption, thus including the possibility of detectable and undetectable levels of data corruption due to selected thresholds. Based on the IMU FMECA, 13 hardware faults where defined, which influence the values stored in the datapool to be used by the processing function. In the analysis we adopt the common single-fault assumption. The faults are constrained to occur at the beginning of a cycle, such that we can analyze how many cycles (ticks) a fault needs to propagate to the function outputs.

Failure modes are the 13 faults. We chose two discrepancies of interest expressed over the output values: degraded (and possibly undetectable) output measurements, and a Boolean data health flag indicating data corruption. The goal was to understand the temporal relationship between faults, the health flag, and degraded (and hence possibly not detected) rate estimations. No multi-mode dynamics are given. The finite-state SMV model has in total 16 Boolean input variables and 84 Boolean state variables. The system diameter of the corresponding reachable state-space is 105; this is the upper bound on the least number of transitions that need to be taken to reach any state from the set of initial states. In total 3488 states are reachable. The model thus has relatively few reachable states, but a considerable depth. The reason is that certain states can

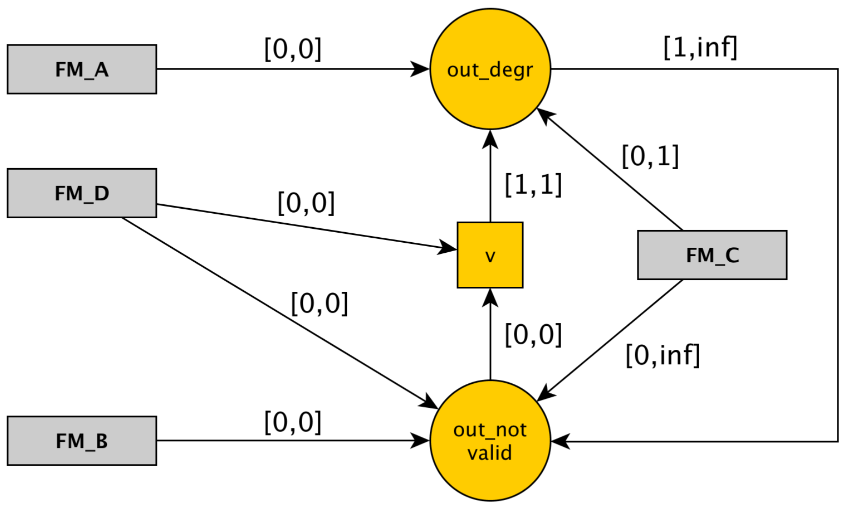


Figure 5.4: Extract of TFPG for the gyroscope processing function. Single-fault assumption is used in the system model, hence only one failure mode can be active at any time. Mode constraints are not shown, as only one mode exists.

only be reached after executing the function several times, and since the function itself also consists of several steps, the overall execution can be quite long. This is important for the performance of model-checkers, which typically decreases with an increasing model depth. The complexity could be avoided by collapsing several computational steps into a single atomic transition, but this on the other hand would make modeling more difficult.

We ran the synthesis and tightening procedures on the problem. We didn't have a clear expectation on the propagation behavior and thus chose synthesis over the manual construction of the TFPG. Figure 5.4 shows part of the synthesized TFPG. Most failure modes have the same edge as "FM_B", and we don't show them here for clarity. The following observations can be made from this result.

- "FM_B" and the failure modes not shown immediately trigger the health monitor and can thus be recognized and adequately handled by the overall IMU processing (if the flag will be used correctly by subsequent functions); the fault doesn't lead to degraded output (but always erroneous output).

- Also "FM_D" immediately triggers the health monitor; furthermore, after exactly one cycle, it will also reach the rate estimation; this is represented by the edge from the virtual `AND` node.
- "FM_C" will affect the rate estimation within one cycle; it might, depending on the fault magnitude, also trigger the monitor, but this is not guaranteed ($t_{max} = +\infty$).
- Finally, "FM_A" immediately results in degraded estimations; the edge from "out_degr" to "out_not_valid" furthermore shows that, after at least one cycle, also the health monitor may trigger, but this is not guaranteed.

These results precisely and concisely show the different propagation behaviors possible in the gyroscope channel processing function, and give formal support to the predictions made in FMECA and FDIR design coverage documents.

Several observations can also be made w.r.t. the adequacy of the framework and performance of the algorithms developed in Chapter 3.

Timing Model The timing model, which differentiates between timed and untimed transitions, comes in handy here. From the analysis point-of-view, the sequence of computation steps is executed instantly. This is of course not true in reality, but is sufficient at this level of abstraction and doesn't matter for the analysis objective. Indeed, precise quantification of individual algorithm steps is difficult as it would require benchmarking all functional steps in the compiled executable, which wasn't available. From the modeling point-of-view, however, it would be very uncomfortable or even impossible to squash all computation steps into one single transition, which the standard LTL view on time, where every transition is a tick, would require.

Semantics of $t_{max} = +\infty$ The semantics of $t_{max} = +\infty$ is very useful here, as in the model we have a situation where the propagation might occur at some point, but might very well also never occur at all. The reason is a non-deterministic transition choice in the system model, that is whether a degraded measurement will trigger internal threshold-based checks. We work with discretized measurement ranges and use this to simulate different magnitudes of degradation; erroneous measurements are assumed always to exceed those threshold, but degraded ones might not. In the TFPG this situation can be represented by $+\infty$. The alternative interpretation that resembles a fairness constraint, i.e. the propagation will always eventually occur but this moment can be delayed by an arbitrary amount of time, would make it impossible to represent this.

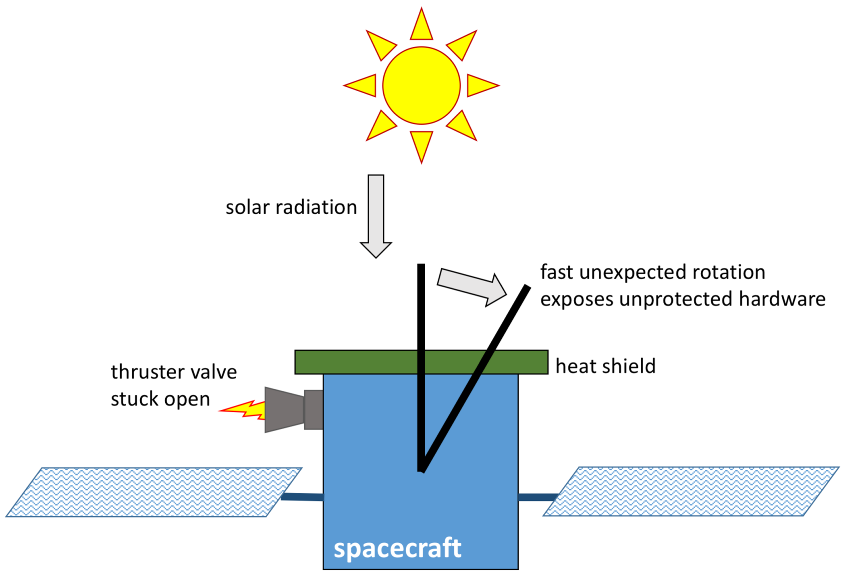
TFPG Simplification Simplification of the synthesized graph is essential for manual inspection. Without it the TFPG would be unreadable for engineers. Indeed already at this level the interpretation of Figure 5.4 is not totally straight-forward.

Performance As for performance, we notice that, on an average desktop computer, synthesis and simplification was completed in 4 seconds by using BDD-based algorithms for minimal cut-set computation. Interestingly, when using IC3 as a back-end engine for cut-sets, graph synthesis takes 210 seconds, which seems to imply that BDD-based reasoning is much better suited to handle the type of model we obtained. For tightening our implementation at the moment can only use IC3 as an engine, because it always reasons on an infinite-state model, as bounds for tmax are not known a-priori. The performance of IC3 seems to decrease with increasing depth of the model, an observation that can be made also in the experiments in Section 3.7. In this specific model the tightening takes 43 minutes, due to the high model depth and the fact that several IC3 calls to search for tight parameter assignments were necessary. The considerable discrepancy with the runtime of the graph synthesis step is due to the fact that graph synthesis is not concerned with timings, and efficient finite-state model-checking can be used for the developed model. This shows the necessity to investigate more efficient tightening procedures, perhaps based on a reduction to finite-state models.

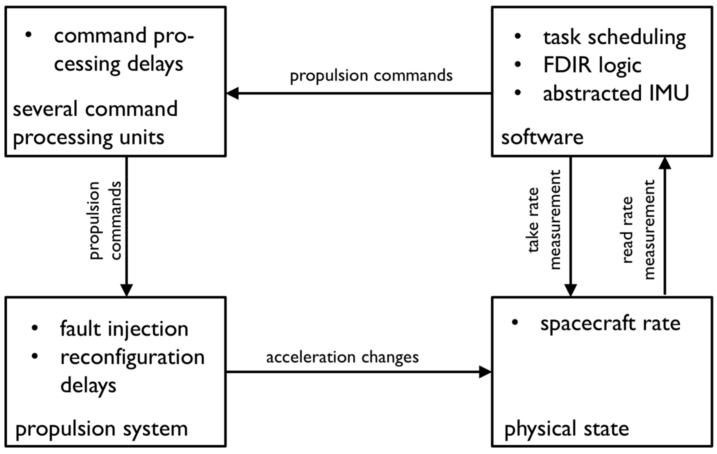
5.2.2 Case Study: Thruster-Valve Stuck

The second case study we performed for TFPG modeling focused on a time-critical propagation scenario. The analysis scope is very different w.r.t. the first case study. We model a flow-control valve of the propulsion system being stuck-open, thus causing a rotation of the spacecraft that might jeopardise the safe zone of the spacecraft attitude. In the case of Solar Orbiter this relates to the requirement to keep the heat-shield pointing towards the Sun. This scenario is illustrated in Figure 5.5a. The analysis is focused on one axis only, which is a reasonable constraint. In Solar Orbiter it can be any thruster that is able to cause the heat-shield to be pointed away from the Sun, in any direction. Note that for other missions similar requirements exist, i.e. keeping the high-gain antennas always pointed to Earth in order to maintain ground contact.

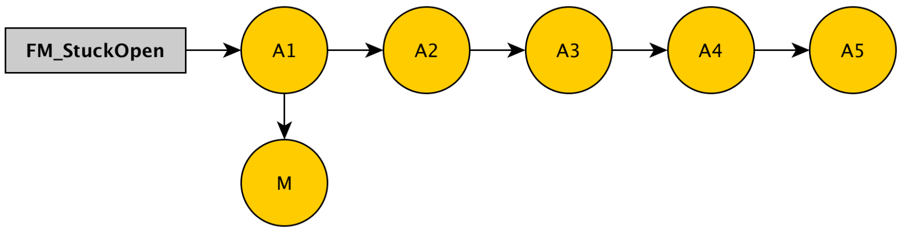
The goal of the case study was to formally validate a timing analysis done by hand, which is the basis for estimating the worst-case spacecraft off-pointing. The chosen scenario was well understood from a discrete perspective: it consists of fault occurrence, detection, and several isolation steps. We developed the TFPG shown in Figure 5.5c; the nodes A1 to



(a) scenario



(b) model layout



(c) TFPG topology (M: monitor; A1-5: off-nominal acceleration phases).

Figure 5.5: Thruster-Valve stuck case study.

A5 are different acceleration phases corresponding to different fault propagation stages, up to the point where fault isolation completely stops the propagation; the node M is a monitor that will be used to trigger the fault isolation. Recovery to an operational state is not included in the analysis; this is acceptable as it is common practice in mission operations to return to operational state only under operator control. The TFPG also contains delay bounds – derived from the documentation – whose precision is in tenths of milliseconds and whose values range from milliseconds to several seconds. For this TFPG we aimed at performing a completeness check, which would confirm the worst-case timing estimates made by engineers.

In Figure 5.5b an abstract overview of the developed model is given[1]. The physical state mainly includes the real-valued spacecraft rotation rate which develops according to an acceleration, which is constant in each propagation phase. The software measures the rate via the IMU and feeds it into the FDIR logic, which consists of several tasks. These tasks are scheduled together with nominal activities. When an off-nominal rate is detected, an alarm triggers, and the FDIR sends several commands to the propulsion system and performs several software operations. The propulsion system includes several valves, in one of which the fault can occur. The spacecraft acceleration is set according to the current configuration of the propulsion system. Delays incur in all parts of the model, from task scheduling to data transmission via communication infrastructure and propulsion system reconfiguration. All basic delays and acceleration constants are modeled in the same detail as found in the documentation, and the model is thus representative for the real physical behavior. Overall the model consists of 7 Boolean and 1 real input variables, as well as 18 Boolean and 5 real state

[1]The actual model can not be presented here due to confidentiality restrictions. However, the model presented in this thesis provides the key elements and concepts that are applicable to many generic spacecraft designs that face similar challenges. Truster-stuck-open is a well-known failure mode that is critical in all missions where high-pointing accuracy is required during time-critical orbital manoeuvres.

variables.

A first completeness check was run with bounded model-checking to have a quick feedback on the delay bound estimates. This check showed that they were not fully accurate with respect to the developed model (the completeness check failed), and the tmax bound on some segments needed to be increased. In other words this meant that the isolation phase took longer in the model than we expected.

As our current implementation of automatic tightening assumes a TFPG that is complete to begin with, we performed a number of manual iterations based on bounded model-checking to identify time bounds that made the completeness test pass. We proceeded in an ad-hoc manner with a mix of linear and binary search steps, based on our knowledge of the problem, and tested several possible values until finding a solution that was precise down to millisecond level. This manual interaction with the model-checker took a couple of hours.

The completeness check on the final TFPG using the IC3 model-checking engine in nuXmv was able to prove the established bounds, with a runtime of 30 minutes on a high-end workstation. This is a relevant result not only from an application perspective, giving feedback on worst-case behavior in a critical scenario, but also from an analysis performance perspective. Indeed they show that it is feasible to analyse very focused but highly accurate propagation problems, and prove respective TFPG properties.

Also in this case study the model traces are rather long. As the model is infinite-state, it is not possible to compute a model diameter. However, the smallest number of steps to reach a state where the last discrepancy in the TFPG is triggered is 90. On the other hand, also here the choice of MTL time model as opposed to LTL is important, also for facilitating modeling but especially for performance reasons. Indeed if we would want to discretize time progression in the model with a sub-millisecond precision, the traces would have several thousands of steps and analysis would be completely unfeasible.

Critical Design Review Even though we were not able to follow up in detail with engineers on the small discrepancies in the propagation delay estimates, the analysis made it possible to raise two issues at the FDIR critical design review of SOLO.

The modeling of this scenario and the TFPG analysis showed that in the documentation it was not clear if the complete duration of the last propagation edge in the TFPG was considered in the worst-case analysis for spacecraft off-pointing. The issue was raised during CDR, and worst-case off-pointing estimates as reported in the documentation were confirmed as accurate.

Furthermore, the spacecraft can be in several operational modes, and thus in principle the TFPG mode labels should cover all of them. For the analysis we followed the documentation and made an explicit assumption of mode-stability for the whole propagation scenario up to the first mode-switching triggered by the FDIR itself. For CDR the issue was raised whether mode-switching is of relevance to the propagation scenario. A corner case was identified by the review panel and confirmed not to influence propagation dynamics, and hence to be covered by FDIR.

5.3 Architectural Propagation Modeling

In this section we try to look at a broader application scope of TFPGs. Whereas in the previous section we studied their application to specific scenarios or components and thus used TFPGs as a type of focused analysis tool, here we try to study whether it is possible and what the challenges would be to use them to model failure propagation from an architectural point-of-view, from unit to subsystem and system levels. These three layers are the usual architectural divisions in spacecraft system design, and also often match the hierarchical organization of FDIR designs, where failures are to be detected, isolated, and recovered at the lowest possible implementation level.

We describe the results of a third case study and discuss possible applications of TFPGs with an architectural focus, as well as ideas for future work:

- in Section 5.3.1 we describe a TFPG modeling case study for SOLO that covers several propagations from unit to subsystem level (AOCS), and describe our findings on using TFPGs as a formalism for propagation modeling across architectural layers; we also describe three contributions to the SOLO FDIR critical design review;
- in Section 5.3.2 we describe the advantages of assessing FDIR design coverage via TFPGs, and briefly outline potential future work on using TFPGs as a framework for FDIR tuning;
- finally, in Section 5.3.3 we argue that TFPGs could be used for diagnostic support during testing/operations in addition to or instead of FMECA tables.

5.3.1 Case Study: IMU to AOCS

The objective of the third case study on applications of TFPGs was to see what kind of topology we would obtain when adopting an architectural perspective, to get hints on overall complexity, and to see how FDIR monitors can be integrated in TFPGs to study FDIR design coverage.

The FAME process suggests to build TFPGs based on the results of previous analyses such as FMEA or FTA. In typical spacecraft design projects

FMEA tables are the central source of information on failure propagation. The ECSS standards indicate FMEA as the primary failure analysis tool, used in all project phases from feasibility analysis (Phase A) up to disposal (Phase F):

> "*The FMEA is an integral part of the design process as one tool to drive the design along the project life cycle.*" ([ECSS-Q-ST-30-02C, 2009], Section 4.1 on General requirements).

FTA instead is recommended as a focused analysis for selected cases:

> "*The supplier shall perform a FTA for [...] selected undesirable events which could have catastrophic, critical or major consequences; [...]*" ([ECSS-Q-ST-40-12C, 2008], Section 5.1.1 on Applicability).

For the case study we thus focus on FMEA analyses, as these are much more common in spacecraft projects. They are organized according to the three architectural layers (unit, subsystem, system): for every component or product at each layer a dedicated FMEA is done and a corresponding table is produced. Typical columns in an FMEA table are (see [ECSS-Q-ST-30-02C, 2009]): ID, item/block, function, failure mode, failure cause, mission phase / operational mode, failure effects, severity, detection method / observable symptoms, and compensating provisions.

We focused on propagations originating from the IMU, reaching subsystem (AOCS) and system (spacecraft) levels, and furthermore focused only on one system mode. Already with this limited focus, propagation modeling turned out to be non-trivial and several challenges could be identified.

Approximately five different design documents that together have several hundreds of pages were instrumental in creating the model. This shows how information related to FDIR is fragmented in current project documentation structures, even when limited to a very narrow scope. Furthermore, also interpreting the documents was not easy, even in collaboration with ESA engineers. This gives an intuition of how difficult it is to manually validate the FDIR design of the *whole* spacecraft and to verify that the overall FDIR design is coherent and covers all possible (and reasonably probable) effects of faults. Similar issues were also encountered in the case studies of Section 5.2.

Failure modes and discrepancies The first issue was to decide what information from the FMEA should be included in the TFPG as failure mode and discrepancy nodes. In other words we asked the question, what are the events that constitute the failure propagation sequences?

The natural candidates here are the failure modes and the corresponding failure effects. In cases where the failure mode is associated to a function, as opposed to a hardware or software component, the failure effect is usually identical to the failure mode, especially at unit level. In our case this was applicable to all unit-level failure modes, and for them we added a single node to the TFPG, declared as TFPG failure mode node. For failure modes two possibilities were considered: relying on the unit-level FMECA done earlier in the project, which contained a detailed set of failure modes, or the consolidated unit-level FMECA produced for CDR, which grouped together unit failure modes that from the FDIR perspective were not distinguished. We chose the first option, which allowed a more accurate integration of monitors in the TFPG and thus a better evaluation of FDIR completeness. The choice also made it possible to identify a failure mode that was not considered in the consolidated FMECA done for CDR. When considering diagnosis applications, this choice would make the diagnosis also more expressive.

At subsystem level the identification was more challenging. Each row in the subsystem table had one associated failure mode, but often more than one failure effect. While the failure modes represented a consolidated list of items (at all levels), the failure effects were less structured. Identifying propagation events thus required interpretation of the informal textual description of what effects a certain failure mode has, in order to extract a consolidated set of propagation events. A specific challenge here was that certain events were mentioned in various parts of different FMEA tables, but the textual description slightly differed, and thus unambiguous identification was not straightforward. Another challenge consisted in the fact that it seemed to be possible to derive distinctive propagation events from both failure mode and failure effect column entries. Whether a failure mode at subsystem level should be modeled in the TFPG as a separate event needed to be assessed by looking at the individual case.

A number of discrepancies were thus derived at subsystem level, declared as `OR` nodes. They were not declared as `AND` nodes, because in FMEA the single-fault assumption is used: failures are not caused by combinations of lower-level events (indeed this cannot be represented in FMEA) but can be traced back to single root events. We assume here that all failure events at subsystem level can be traced back to events at unit level, and thus don't introduce dedicated TFPG failure mode nodes at subsystem level.

Finally, we also wanted to investigate the integration of monitors into the TFPG, and thus added the monitors established by the proposed FDIR design. Two categories of monitors are commonly used, also in SOLO: standard monitors (SMON), which are simple Boolean expressions over raw or computed observable signals, and functional monitors (FMON), which are composed of a set of standard monitors. Functional monitors can have either AND semantics, or OR semantics; when using AND semantics, the monitor triggers when all associated standard monitors have triggered, whereas for OR semantics just one SMON needs to trigger. Standard monitors can be seen as fault symptoms, and functional monitors, which are used to trigger recoveries, represent slightly more complex diagnoses and are from an architectural point-of-view comparable to the alarms we require diagnosers to produce in Chapter 4.

The design of these monitors thus perfectly matches the notion of `OR` and `AND` discrepancies in TFPGs. Just as the nodes in TFPGs, these monitors are conditional on the occurrence of a fault, otherwise false alarms would be possible. Furthermore we note how the standard monitors match the notion of a discrepancy being defined by a Boolean expression over system variables, whereas functional monitors match the notion of virtual discrepancies that are defined based on other discrepancies that have edges towards it.

Even though in this case study we didn't create a system model to compare the TFPG against, it became clear that a considerable difficulty would be in defining certain TFPG nodes. It seems to be pretty straightforward at the unit level, in our case with clear effects on the IMU hardware. However, at the subsystem level the FMEA uses terms such as "fast", "slow", and "high", without a formal definition being available in the project documentation. For a precise definition, which we would need for TFPG validation or synthesis, additional interaction with engineers would be necessary. Beyond TFPG validation, such information would make validation of the FDIR more effective.

Qualitative Edges The next question to consider was how to connect the nodes. We first focus on the qualitative graph topology. Two approaches to link failure mode and discrepancy nodes (excluding monitor nodes for a moment) were identified based on the FMEA tables: "forward linking" by considering the columns for failure effects at the higher architectural level (this was chosen for the case study), and “backward linking” via the possible-cause column. By “forward” we mean following the same direction as the propagation, and by “backward” the opposite direction as the propagation.

For forward linking we focus on one FMEA table row and look at the prediction of failure effects at the next level. These should match with the failure effects of some failure mode at the higher level. Relating table rows was possible this way but not fully straightforward, due to the less structured content of failure effect table cells. What we were able to do with this approach was to match rows of different FMEA levels. However, since the FMEA rows at subsystem level in our use case correspond to more than one node in the TFPG, additional interpretation of the nature of individual events and interaction with engineers were necessary to establish the exact temporal ordering via TFPG edges. This additional knowledge allowed us to create a clearer propagation model compared to how the FMEA tables represent propagation.

A second approach to relate FMEA rows is, in principle, the possible-cause column, which should enable a backward linking. However we found this to be conflicting with our choice of forward linking, because in the available tables this backward perspective had an implicit assumption of fault isolation. It indicated failures at a lower level that are not detectable or recoverable at that level, thus excluding all failure modes for which monitors and recoveries were defined there.

It seems thus that, in fact, two different implicit and possibly conflicting propagation models are present in the FMEA tables: one where FDIR fails or is not executed and the failure thus propagates further to the next level (forward linking), and one where FDIR cannot, by design, prevent a propagation (backward linking).

Edges towards monitor discrepancies were easier to establish, also benefiting from the preceding modeling steps. In “Failure Effect Summary List” (FESL) tables the standard and functional monitors are associated with the consolidated failure modes. Based on this and the precise definitions of the monitors it was possible to establish the edges from TFPG nodes to SMON nodes; edges from SMON to FMON nodes directly followed from the definition of FMON monitors.

Edge Constraints In this case study we focused only on one mode. However, in FMEA tables effects of failure modes can be bound to certain operational modes. This information can be used to establish mode labels in the TFPG.

We also didn't model refined time bounds on the propagations, but set them to default values of $t_{min} = 0$ and $t_{max} = +\infty$. These values represent the extreme cases of propagation delays. In the case of monitors, immediate propagation means that the monitor triggers immediately as soon as the associated condition becomes true (best case), and infinite delay means that the monitor will never trigger (false negative, worst case). In the case of unmonitored failure effects, the interpretation is swapped. Immediate propagation is the worst case, because there is zero time to react, and infinite delay is the best case in the sense that the feared event will never occur.

TFPG Properties The structure of the TFPG that we developed for this case study is shown in Figure 5.6. It has the following properties:

- 13 failure modes (IMU-gyro failure modes)
- 19 discrepancies (4 subsystem/system-level failure mode/effect items, 9 standard monitors, 6 functional monitors)
- 13 edges from failure modes to failure effect discrepancies

- 22 edges from failure modes and discrepancies to standard monitors
- 9 edges from standard monitors to functional monitors
- 3 edges among failure modes/effects encoded as discrepancies

For this example it can be seen that the failure modes and the monitors constitute the biggest sets of nodes, while the unobservable failure effects are significantly fewer in number. This shows that gyroscope failures have a limited number of effects on the system in terms of feared events, and that all but the top-level one are captured by monitors.

Results The result of this case study is a TFPG that models failure propagations rooting in the IMU more clearly and in a more integrated way than the corresponding FMEA documentation. We showed a way to integrate standard and functional monitors in the TFPG. This makes a more formal assessment of FDIR design coverage possible (discussed in Section 5.3.2) and also potentially better supports, compared to FMEA, diagnostic activities during later project phases (see Section 5.3.3).

In the TFPG it can be seen that no connections are made between SMON nodes, as no information was available to allow such modeling. It would be interesting though to see whether such connections could be established, and whether that would lead to different (better) FMON definitions.

Timings were not considered in the TFPG. This would have required more interaction with project engineers in order to learn how to derive this information from the documentation. The information might not even be available as it is usually not considered during FMEA. Compared to FMEA the TFPG makes it clear that, without further information and from a formal point-of-view, we need to assume propagation can be instantaneous

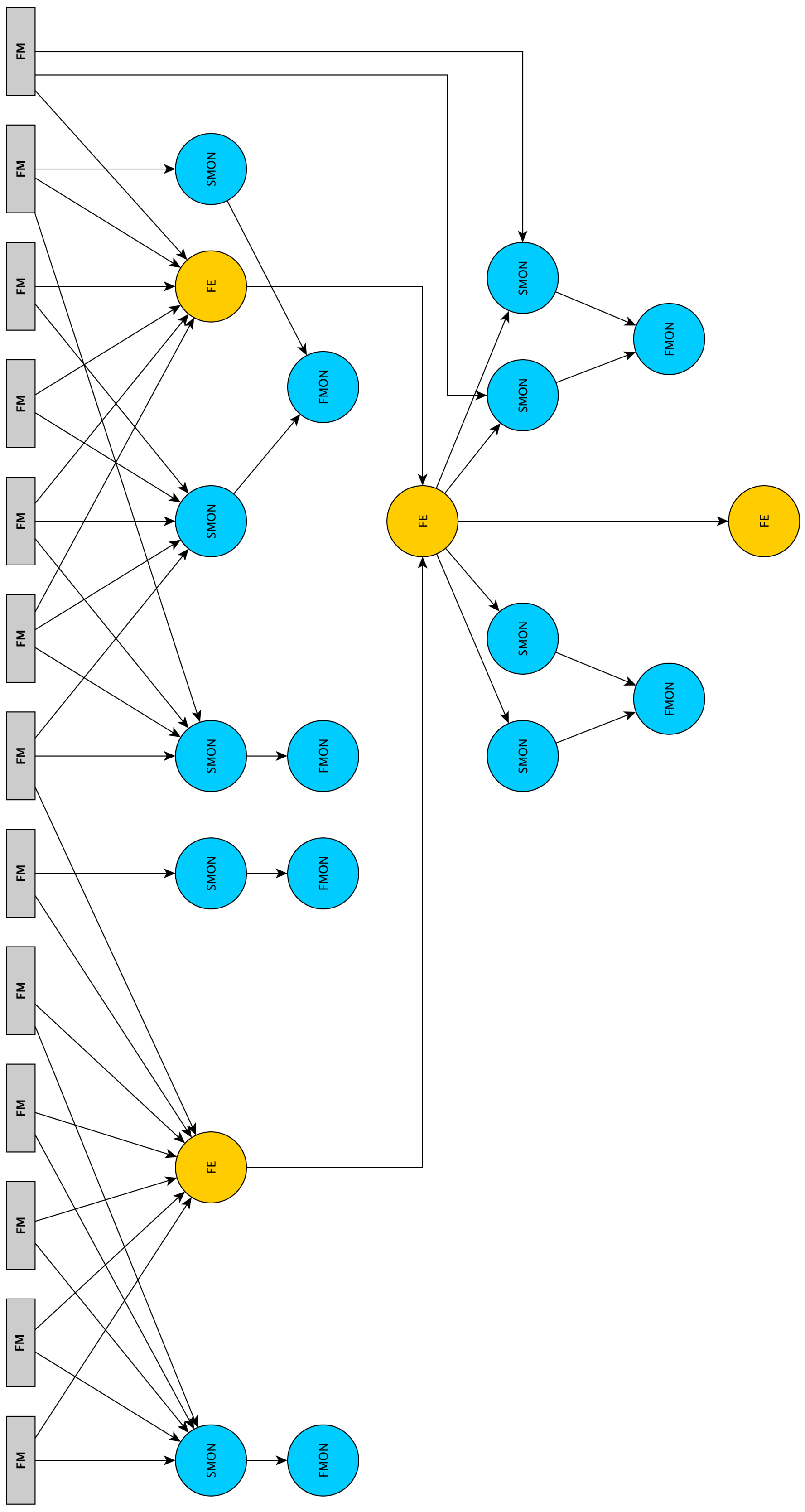


Figure 5.6: TFPG of the IMU-to-AOCS case study. FM: failure mode; FE: (unobservable) failure effect; SMON: standard monitor; FMON: functional monitor.

or might never occur at all, forcing engineers to be explicit about timing aspects.

With respect to delay bound modeling, an interesting follow-up case study would be to try using TFPG models with such maximally permissive time bounds for a specific task that FMEA is used for. The ECSS standards require FMEA to provide the following result during Phase B:

> "*Identification of failures requiring failure detection and recovery action in a time interval greater than the time to an irreversible consequence [...].*" ([ECSS-Q-ST-30-02C, 2009], Section 6.4, item e.1).

The clear identification and definition of propagation events and links gives a more formal framework to investigate this question, and by integrating also monitors in the TFPG allows to directly compare worst-case (fastest) propagation times to feared events and worst-case (slowest) triggering delay of potential monitors. The delay estimates might not even have to be precise down to milliseconds to identify potentially critical scenarios, and likewise to identify scenarios where further analysis is not necessary.

The qualitative information contained in the graph of Figure 5.6, disregarding timing and mode information, could also be represented in tabular format, similar to FMEA. It is however arguable that the result is an FMEA table, as all nodes that are not roots of the graph would need to be declared as failure mode nodes, which is not intuitive from a methodological point-of-view. In plain tabular format it would also be impossible to represent AND semantics, which might be used by FMON monitors, or in the case of multiple-fault modeling. Fault trees could also be used to represent the qualitative relationships between nodes, and would also support AND semantics. Indeed, this is the basic idea on which our TFPG synthesis approach is based. But depending on how exactly the fault trees are built, this might result in 7 to 19 different trees, making it more difficult to obtain a global picture on possible failure scenarios and the relationship between monitors and failure effects.

Critical Design Review The modeling efforts described in this section raised several questions that were forwarded to the CDR panel.

A first question regarded the rationale behind the hierarchical placement of two failure modes, as their direct effects, according to FMECA, influences the whole system. This issue was identified as our modeling goal was, based on failure effects, to link unit and subsystem levels. Thus it was not fully clear how many monitor/response safety layers were in place to prevent propagation to system level. It was clarified during the review that while the failure effects influenced the whole system, the failure modes were placed at subsystem level because detection, isolation, and recovery involves only that subsystem. Hierarchical placement in the FMEA analyses thus is not always oriented on the “architectural perimeter” of the fault effect, but is sometimes rather based on the architectural structure of FDIR.

To identify unit-level failure modes we were using the detailed FMECA at unit-level instead of the consolidated one developed for CDR. The consolidated FMECA results are the ones used to assess FDIR design coverage, and summarize, especially at unit-level, several failure modes into combined ones. However, during our analysis we discovered one failure mode at unit-level that, according to the unit-level FMECA, was not detectable at unit-level. In the design coverage tables however all consolidated unit-level failure modes are detectable at unit-level with the proposed monitors. It was confirmed during the review that in fact detection of the identified failure mode at unit-level is possible, and that the FMECA table was incomplete. The information on what monitor would handle the failure mode was added to the documentation. In addition, also the association with subsystem-level monitors was clarified.

Finally, by trying to model the IMU-to-AOCS TFPG we identified an ambiguity on the exact ordering of propagation events; this situation was clarified as well during the review. The problem is that propagation is supposed to connect failure effects at various levels. As shown in the case study, identifying those events and connecting them is not always intuitive.

5.3.2 Assessment and tuning of FDIR design coverage

The general problem that drove the TFPG case studies was the validation of FDIR design coverage. This meant to check whether every failure mode is caught by a monitor at the same architectural level to guarantee its detection, what propagations to higher levels are possible and whether monitors at the higher level were in place as a fall-back detection mechanism.

FDIR design coverage is usually based on "Failure Effect Summary List" (FESL) tables, of which a simplified example is given in Table 5.1. For each architectural component, from units to subsystems, such a table is made. The first set of columns describes the failure mode and is derived directly from the FMEA tables; the example contains the failure mode (FM), function/item (FUN/ITM) which the failure mode is associated to, failure effect (FE), possible cause (PC), and textual description of observable symptoms (OBS). The second set of columns (M1 to M3) are the monitors; selected cells (marked with X) indicate that the monitor of that column is expected to trigger following the failure mode of that row. The third set of columns shows the recoveries; marks associate recoveries with failure modes they are supposed to isolate and possibly recover the system from. Recovery information is integrated in FESL tables only for convenience, as the precise association between monitors and recoveries is described elsewhere. However, these tables are the main source of information for associating failure modes and monitors.

FMEA Information					Monitors			Recoveries	
FUN/ITM	FM	FE	PC	OBS	M1	M2	M3	R1	R2
...					X	X		X	
...						X	X	X	X

Table 5.1: Simplified example of Failure Effect Summary List (FESL) table layout. Multiple monitors can be associated to a failure mode row. Recoveries are included for convenience, their precise association with monitors is specified elsewhere.

Based on these tables, engineers reviewing and assessing FDIR design coverage are interested in the following two main questions:

1. How exactly are the monitors related to the failure mode row?
2. What happens if FDIR at this level fails in terms of propagation?

The experience with SOLO shows that both questions are not trivial when working with typical FMEA tables. Each failure mode may contain more than one distinct event which have to be identified by interpreting the textual informal description in the row cells. Furthermore, if multiple monitors are assigned to the row, then there is no information in the table on what exact event each monitor is associated with. With TFPGs, instead, it is possible to clearly describe what the events of interest are and how we assume them to be related in a temporal sense among themselves and w.r.t. the monitors. From a purely qualitative point-of-view, coverage can thus be assessed by checking what monitors are reachable from every (unobserved) failure event. With the delay bounds TFPGs give also additional information not contained at all in FESL tables, and allow to compare fastest propagation time to the next event against the slowest propagation time towards the monitor (upper detection delay bound).

TFPGs also clearly state the worst-case when no timing information is given, by setting the t_{max} delay towards monitors to $+\infty$, which means the monitor might never trigger at all. FESL tables however seem to have a more optimistic view and assume that the monitors will always trigger in response to the failure mode or one of the failure effects.

The second important question is also not straightforward to answer with FESL tables, being directly derived from FMEA tables: What will happen in terms of propagation when the FDIR fails to detect a failure mode or to recover from it, and is a fall-back monitor/response pair in place to capture the propagation? In the case study described in Section 5.3.1 we showed that propagations between different architectural layers can be represented in TFPGs, thus connecting local information into a global propagation model. This then allows to assess how many and which failure events with associated monitors a propagation has to go through before reaching a point where no further monitors (and recoveries) exist.

The experience of the case study showed that precise modeling of failure propagation and relationship of failures and monitors is difficult based on commonly used project documentation, due to the fact that relevant information is scattered over several documents. On the other hand, TFPG modeling offers a clear and concise picture on failure propagation. It would thus be interesting to see whether the documentation typically used to assess FDIR coverage could be generated from TFPGs, resulting in documents that are more consistent and perhaps much more concise.

FDIR Tuning It does seem thus that the task of checking FDIR design coverage can be reduced to the task of building a TFPG from the FMEA and FESL tables. The result is a unified model relating failure modes, effects, and monitors. Beyond its application to FDIR design validation, it should also be possible to use such TFPGs for tuning the overall FDIR implementation as follows.

A usual first task during recoveries is, right after the associated monitor triggers, to disable other monitors that might trigger other interfering recoveries. Choosing what exact monitors should be disabled is challenging and could be supported by TFPGs. A possible algorithm could be to compute all possible nodes that can reach a monitor M of interest, and then check what other monitors are reachable as well from those nodes. This gives an over-approximation of what monitors can trigger too after M triggers, and the associated recoveries can be checked for compatibility with the recovery associated with M.

A second possible way to use TFPGs for tuning FDIR exploits timing information on the edges. Fault identification, as opposed to simple detection, is sometimes achieved by waiting "long enough" for the "right" monitor to trigger. Figure 5.7 shows a small but realistic example. Three failures are possible: the IMU can break, the star tracker (STR), or the bus that connects both to the central computer.

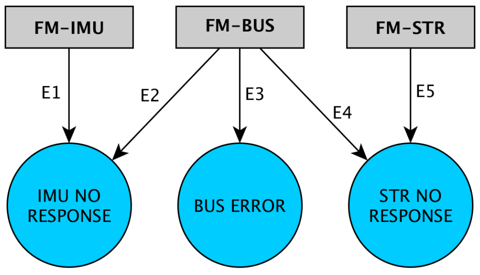


Figure 5.7: Example for filter tuning. The filter values for the IMU and STR monitors need to be chosen such that for each failure mode the corresponding monitor will trigger first.

The goal is to trigger, for each failure, a corresponding recovery. The default approach in most FDIR implementations to launch recoveries is to execute the one associated to the first monitor that triggers. Assuming single-faults, for both IMU and STR failures only the appropriate monitor will trigger. However, with a bus failure it might happen that the IMU or STR monitors will trigger before the BUS monitor, thus causing the wrong recovery to be executed. What can be done is to delay, in the FDIR implementation, the triggering of IMU and STR monitors by a value δ such that, if FM-BUS occurs, the BUS monitor will trigger first. More precisely we require that $tmax(E3) < tmin(E2) + \delta_{IMU}$ and $tmax(E3) < tmin(E4) + \delta_{STR}$. Naturally this would also mean to delay the recoveries for IMU and STR failures – but not the propagation towards the SMON in the TFPG.

Finally, a third possible application of TFPGs for FDIR tuning is to tune thresholds and detection filters which are often used in monitor definitions. Thresholds are commonly used to detect the presence of faults through abnormal readings; filters, which can be counters or timers, are used to specify how long or how many times the threshold violation has to persist such that a fault can be confirmed and spurious triggerings be excluded.

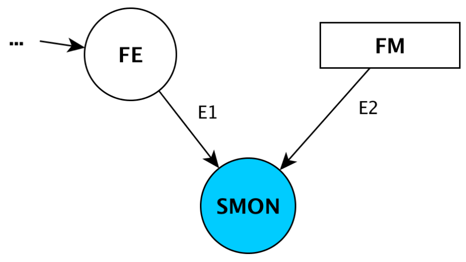


Figure 5.8: Example use case for monitor threshold and detection filter tuning. Values need to be found that optimize the monitor's reactivity without introducing false alarms.

An example is shown in Figure 5.8. Assume SMON is defined by an expression $\phi := v \geq \theta$ (where v is a variable being measured and θ a threshold dividing nominal and off-nominal values), and by a filter value f that indicates how many times in a row the sampling of v has to result in ϕ evaluating to *true*. The analysis framework described in Chapter 3, specifically the TFPG completeness check, can then be used to validate the choices for θ and f. For this a model is needed that describes the dynamic nominal and faulty behavior of the system of reference. If only the threshold and filter choices for SMON shall be evaluated, only the proof obligations $\psi_{\mathtt{OR}\cdot A}(SMON, \Gamma)$ and $\psi_{\mathtt{OR}\cdot B}(SMON, \Gamma)$ need to be checked. An automatic procedure to optimize θ and f can be obtained by lifting the validation problem with fixed values to a parameter synthesis problem: θ and f parameterize the implementation of SMON inside the model, and the t_{max} values of the two edges are used as optimization criteria – lower t_{max} values mean better worst-case detection delays.

5.3.3 Diagnostic support for testing and operations

The analysis of project documentation and the development of the TFPG described in Section 5.3.1 and interaction with ESA engineers also raised the issue of diagnostic support via FMEA/FMECA results. Even though we didn't have the opportunity to investigate this topic further, we identified three promising directions for future work and case studies.

The ECSS Standard on FMEA/FMECA states, referring to Phase D (production or ground qualification testing), that:

> *The FMEA/FMECA shall be utilized as a diagnostic tool in order to support the failure diagnosis during the qualification and the elimination of potential failures.*
> (ECSS-Q-ST-30-02C, Section 6.6.)

Furthermore, referring to Phase E (utilization):

> *The FMEA/FMECA performed at system level in phase C/D shall be utilized as support to diagnostic activities (in-flight and on ground) in order to support system maintenance and restoring.*
> (ECSS-Q-ST-30-02C, Section 6.7.)

Even using the FMEA results for very thorough diagnostic troubleshooting as required by ECSS, in practice it is sometimes difficult or even impossible to identify root causes of events. This then causes healthy components to be marked as unhealthy and corresponding redundancies to be activated, just because it is not clear what exactly caused a specific issue and because it cannot be risked to use a possibly compromised unit or to rely on a compromised fault-detection mechanism. In fact the problem doesn't seem to be as much one of fault detection than one of precisely identifying the faulty component or item, hence making it impossible to perform more fine-grained recoveries. Furthermore, in missions that have to guarantee double-fault-tolerance this can also result in adding additional redundant units or components to the system, which increases spacecraft weight, power consumption, system complexity, and costs in general. This could in principle be avoided by a better understanding of how various faults are connected and how they influence the available monitors. Improving diagnosability is thus a very important issue in practice, both from an operational perspective (unnecessarily losing redundancy during operations results in reduced capability to deal with subsequent faults) and from an economical perspective (unnecessarily adding redundancy in the design increases various costs).

Our experience during the case studies showed that TFPGs can be used to represent failure propagations and association of monitors with failure events more precisely than FMEA and FESL tables. The conjecture is that the higher precision and formality in propagation modeling, and possibly the addition of timing information where known, can help to produce better diagnosis results.

As a first follow-up activity it would thus be interesting to see whether formal propagation modeling with TFPGs would alleviate the kind of diagnosis problems mentioned above, either by being able to associate more accurate diagnoses to monitors that trigger recoveries online, or when being used as a support for manual diagnosis or troubleshooting in addition or instead of FMEA tables.

A second question that should be investigated is to use the diagnosis approach based on discretized TFPGs as described in Section 5.1. The case study in FAME was rather focused, and it would be interesting to see whether applying the diagnoser synthesis on architectural TFPGs that span the whole system would result in better diagnosability as compared to the diagnoses represented by functional monitors. As observables the standard monitors and possibly other observable symptoms can be used, and the diagnoser alarms would be computed based on those observations and by reasoning on the TFPG structure and the delays between key events.

5.4 Summary

In this chapter we described several ways in which TFPGs can be applied in a project setting. In Section 5.1 a translation of TFPGs to the SMV language is presented with the goal to enable, on top of it, diagnoser synthesis in the sense of [Bozzano et al., 2015c]. This technique has the potential to improve diagnostic accuracy, as discussed in Section 5.3.3.

In Section 5.2 we describe two case studies on TFPG validation and synthesis, based on the ESA Solar Orbiter project. One of them focuses on a software component, and another analyzes a critical propagation scenario at a higher abstraction level. They show the adequacy of the framework developed in Chapter 3 as well as the performance on a detailed and realistic model. Contributions to the SOLO FDIR critical design review are described.

In Section 5.3 a third case study is presented on modeling propagation across architectural layers, as well as the integration of FDIR monitors in the resulting propagation model. The usefulness of this modeling approach for FDIR design coverage and diagnostic support is demonstrated. The TFPG we obtained is comparable in complexity to the experiments in Section 3.7, which confirms that those are based on realistic problems. Also based on this case study contributions to the SOLO CDR were made.

During the research stay at ESTEC we didn't perform case studies for diagnosability due to a lack of time, but instead focused on potential applications of TFPGs to maximize our findings on that topic. However, the general experience and interaction with engineers also clearly showed the importance of diagnosability analysis as described in Chapter 4, especially for enabling effective redundancy management through more precise fault localization.

When using TFPGs for diagnosis we make the essential assumption of simple observable discrepancies (standard monitors) being available. But what if such monitors are not given and it is not clear how to derive them, such that specific diagnostic objectives can be met? Do we even have enough information to implement such monitors, given a set of observables and knowledge of the system behavior? What if we need to diagnose something more complex than a Boolean expression over state variables, such as an anomalous behavior? This is the challenge that the framework and algorithms described in Chapter 4 on diagnosability try to deal with.

The present dissertation focuses on propagation modeling and diagnosis tasks. Working with actual project documentation on FDIR designs we were also able to identify several opportunities for future work specifically dealing with recovery aspects.

First we want to point out that the developed framework for TFPG analysis can be used to validate the effectiveness of fault isolation, that is the ability to stop faults from propagating. Indeed this has been done in the case study of Section 5.2.2. In some sense this is done by duality, as TFPGs describe propagations that are indeed possible and cannot be isolated in all circumstances.

Beyond fault isolation, there is a clear need for a framework to formulate recovery requirements, similar in spirit to the specification framework used in Chapter 4 for expressing diagnosis requirements. Recovery requirements are usually formulated in natural informal language or only implicitly, and a clear pattern language can help to streamline and formalize them. Based on such formal patterns it is important to investigate the issues of validating a given recovery strategy, checking the existence of one and automatically computing it. For this it could be considered to rely on planning frameworks, as suggested in FAME.

Chapter 6

Conclusion

In this dissertation we advanced the state-of-the-art in formal failure analysis. In many modern engineering systems safety and availability are critical properties, and a thorough assessment of how faults impact a system – and thus potentially compromise those properties – is fundamental.

Timed Failure Propagation Analysis The first technique we investigated are Timed Failure Propagation Graphs. TFPGs are well-suited to model how faults affect a system over time and have distinctive advantages over traditional modeling techniques. A framework was developed to treat TFPGs as abstractions of transition systems. Such models can be used to describe the dynamic behavior of a system, including behavior in presence of faults. Based on this framework, algorithms are developed for validating TFPGs with respect to and synthesizing TFPGs from transition systems. The techniques for TFPG validation, where a TFPG is given, can thus be used by an engineer to validate the assumptions on failure propagation encoded in the TFPG. The techniques for TFPG synthesis can be used when such assumptions cannot be made. The implementation core is based on symbolic model-checking, which is one of the main technologies in formal verification. Experimental results show promising performance and thus the potential of the approach for industrial application.

Diagnosability Analysis The second technique for failure analysis that was investigated is diagnosability analysis. The goal here is to study whether failures affect the available observables in a way that makes diagnosis possible within a specific time frame. The contribution builds on epistemic definitions of diagnosability that are extended with the notion of operational context. Verification of diagnosability under these definitions is reduced to checking the existence of critical pairs; necessity and sufficiency properties of this approach are analyzed. The implementation relies on model-checking of the twin-plant, which is used to produce indistinguishable pairs of traces. The synthesis problem for diagnosability is also addressed: the goal here is to identify subsets of the possible observables that still guarantee diagnosability but also optimize the total cost. We propose a synthesis algorithm that relies on a parameterized twin-plant and reduction of the problem to parameter synthesis; experimental results show the competitive performance w.r.t. state-of-the-art approaches.

Industrial Application In the FAME project a method to translate TFPGs to the SMV language was developed, with the goal of performing diagnosability analysis and synthesis of diagnosers on top of it. The approach was validated by an industrial partner in a case study for the ExoMars Orbiter.

During a research stay at ESTEC three case studies were developed based on a spacecraft currently under development. The case studies showed the adequacy of the framework and the performance of the implementation. They furthermore show the advantages of TFPGs in formally and concisely representing the relationship between failure propagations and monitors, which makes it considerably easier to review an FDIR design. The benefits of TFPG modeling were also demonstrated through several contributions to the critical design review of the analyzed mission.

During the review of FDIR documentation at ESA also the high practical relevance of diagnosability analysis became evident. A big problem in fact is that often failure identification can be done only to a degree that requires reconfiguration of larger parts of the spacecraft, without the possibility to precisely locate the source of the failure. By worst-case reasoning this results in declaring potentially healthy components as unhealthy. Diagnosability analysis can alleviate the situation by proving that either the precise diagnosis is actually possible, or by showing that indeed not enough observables are available.

Future Work Many promising directions of future work exist for all topics mentioned in this dissertation. Several are discussed in the respective chapters, and here we would like to point out two issues concerning a tighter integration of TFPGs and diagnosability of transition systems.

First of all we want to extend the diagnosability framework to use more expressive encodings of time as they are also used for our TFPG approach. This would make it possible to reuse the same system models for both analyses, which is instrumental to design consistency.

Based on this we want to develop TFPGs containing only feared events, and design the monitors to be integrated with diagnoser synthesis. Integration of failure propagation dynamics and monitoring facilities not only enables a more global analysis, but is also very useful for supporting FDIR design review as shown in the case studies. For this we need to investigate the issue of zero-recall diagnosability and diagnoser synthesis to identify monitor implementations. The standard monitors typically used in FDIR design can be described as zero-recall diagnosers, as they are evaluated considering just the current state of all observables. As in current industrial FDIR design, such monitors probably need to focus on simpler conditions, as diagnosis of more complex conditions is unlikely to be feasible by observing single snapshots of observations.

Finally, development of a specification framework for recovery requirements is necessary. Based on this we will investigate definitions of recoverability for transition systems or possibly dense-time frameworks, as well as corresponding algorithms to automatically compute recovery procedures.

Bibliography

S. Abdelwahed, G. Karsai, N. Mahadevan, and S.C. Ofsthun. Practical implementation of diagnosis systems using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58 (2):240–247, 2009.

Karine Altisen, Franck Cassez, and Stavros Tripakis. Monitoring and Fault-Diagnosis with Digital Clocks. In *International Conference on Application of Concurrency to System Design*. IEEE Computer Society, 2006.

Rajeev Alur and Thomas A Henzinger. Real-time logics: complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.

Étienne André, Thomas Chatain, Laurent Fribourg, and Emmanuelle Encrenaz. An Inverse Method for Parametric Timed Automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, 2009.

AUTOGEF. AUTOGEF Project Web Page: `https://es.fbk.eu/projects/autogef_main`, 2016. URL `https://es.fbk.eu/projects/autogef_main`.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

Anupa Bajwa and Adam Sweet. The Livingstone model of a main propulsion system. In *Proceedings of the IEEE Aerospace Conference*, pages 63–74, 2003.

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

Mehdi Bayoudh and Louise Travé-Massuyès. Diagnosability Analysis of Hybrid Systems Cast in a Discrete Event Framework. *Discrete Event Dynamic Systems*, 24(3):309–338, 2014.

A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model-checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer, 1999.

Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science*, 66 (2):160–177, 2002.

S Biswas, D Sarkar, S Mukhopadhyay, and A Patra. Diagnosability Analysis of Real Time Hybrid Systems. In *IEEE International Conference on Industrial Technology*, pages 104–109. IEEE, 2006.

Santosh Biswas, Dipankar Sarkar, Siddhartha Mukhopadhyay, and Amit Patra. Fairness of Transitions in Diagnosability of Discrete Event Systems. *Discrete Event Dynamic Systems*, 20(3):349–376, 2010.

B Bittner, M Bozzano, Alessandro Cimatti, M Gario, and Alberto Griggio. Towards pareto-optimal parameter synthesis for monotonic cost functions. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 23–30. FMCAD Inc, 2014a.

Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, and Xavier Olive. Symbolic Synthesis of Observability Requirements for Diagnosability. In *AAAI Conference on Artificial Intelligence*, 2012.

Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, Regis De Ferluc, Marco Gario, Andrea Guiotto, and Yuri Yushtein. An Integrated Process for FDIR Design in Aerospace. In *Model-Based Safety and Assessment*, pages 82–95. Springer, 2014b.

Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xSAP Safety Analysis Platform. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 533–539. Springer, 2016a.

Benjamin Bittner, Marco Bozzano, and Alessandro Cimatti. Automated synthesis of timed failure propagation graphs. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 972–978, 2016b.

Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, and Gianni Zampedri. Automated verification and tightening of failure propagation models. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*, 2016c.

Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - An Optimizing SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.

Andrea Bobbio, Luigi Portinale, Michele Minichino, and Ester Ciancamerla. Improving the analysis of dependable systems by mapping fault trees into bayesian networks. *Reliability Engineering & System Safety*, 71(3):249–260, 2001.

Abderraouf Boussif and Mohamed Ghazel. Diagnosability Analysis of Input/Output Discrete-Event Systems Using Model-Checking. *IFAC-PapersOnLine*, 48(7):71–78, 2015.

M Bozzano, A Cimatti, A Guiotto, A Martelli, M Roveri, A Tchaltsev, and Y Yushtein. On-board Autonomy via Symbolic Model-based Reasoning. In *ESA Workshop on Advanced Space Technologies for Robotics and Automation*, 2008.

M. Bozzano, A. Cimatti, M. Roveri, and A. Tchaltsev. A Comprehensive Approach to On-Board Autonomy Verification and Validation. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2011.

M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta. Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *Proc. CAV 2015*, pages 518–535, 2015a.

Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 162–176, 2007.

Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. The COMPASS approach: Correctness, modelling and performability of aerospace systems. In *Computer Safety, Reliability, and Security*, pages 173–186. Springer, 2009.

Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal Design of Fault Detection and Identification Components Using Temporal Epistemic Logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 326–340, 2014a.

Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, and Stefano Tonetta. Formal Safety Assessment via Contract-Based Design. In *International Symposium on Automated Technology for Verification and Analysis*, pages 81–97. Springer, 2014b.

Marco Bozzano, Alessandro Cimatti, Marco Gario, and Andrea Micheli. Smt-based validation of timed failure propagation graphs. In *Twenty-ninth AAAI Conference on Artificial Intelligence*, 2015b.

Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal Design of Asynchronous FDI Components using Temporal Epistemic Logic. *Logical Methods in Computer Science*, 2015c.

Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Cristian Mattarei. Efficient Anytime Techniques for Model-Based Safety Analysis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 603–621, 2015d.

Aaron R Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

Davide Bresolin and Marta Capiluppi. A Game-theoretic Approach to Fault Diagnosis and Identification of Hybrid Systems. *Theoretical Computer Science*, 493:15–29, 2013.

L.B. Briones, A. Lazovik, and P. Dague. Optimal Observability for Diagnosability. In *International Workshop on Principles of Diagnosis*, 2008.

Maria Paola Cabasino, Alessandro Giua, and Carla Seatzu. Diagnosability of Bounded Petri Nets. In *IEEE Conference on Decision and Control*, pages 1254–1260. IEEE, 2009.

Maria Paola Cabasino, Alessandro Giua, Stéphane Lafortune, and Carla Seatzu. A New Approach for Diagnosability Analysis of Petri Nets Using Verifier Nets. *IEEE Transactions on Automatic Control*, 57(12):3104–3117, 2012.

Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems, Second Edition.* Springer, 2008.

F. Cassez, S. Tripakis, and K. Altisen. Sensor Minimization Problems with Static or Dynamic Observers for Fault Diagnosis. In *International Conference on Application of Concurrency to System Design*, pages 90–99. IEEE, 2007.

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model-checker. In *Computer Aided Verification*, pages 334–342. Springer, 2014.

Sébastien Chédor, Christophe Morvan, Sophie Pinchinat, and Hervé Marchand. Diagnosis and Opacity Problems for Infinite State Systems Modeled by Recursive Tile Systems. *Discrete Event Dynamic Systems*, 2014.

Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *Computer Aided Verification*, pages 277–293. Springer, 2012.

Alessandro Cimatti, Charles Pecheur, and Roberto Cavada. Formal Verification of Diagnosability via Symbolic Model-Checking. In *International Joint Conference on Artificial Intelligence*, pages 363–369, 2003.

Alessandro Cimatti, Luigi Palopoli, and Yusi Ramadian. Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In *Real-Time Systems Symposium, 2008*, pages 80–89. IEEE, 2008.

Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. SMT-Based Verification of Hybrid Systems. In *AAAI Conference on Artificial Intelligence*, 2012.

Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter Synthesis with IC3. In *International Conference on Formal Methods in Computer-Aided Design*, pages 165–168, 2013a.

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013b.

Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 Modulo Theories via Implicit Predicate Abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.

Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. HyComp: An SMT-based Model-checker for Hybrid Systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 52–67. Springer, 2015a.

Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. HRELTL: A

Temporal Logic for Hybrid Systems. *Information and Computation*, 245: 54–71, 2015b.

Alessandro Cimatti, Marco Gario, and Stefano Tonetta. A lazy approach to temporal epistemic logic model checking. In *International Conference on Autonomous Agents & Multiagent Systems*, pages 1218–1226. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

Koen Claessen and Niklas Sörensson. A Liveness Checking Algorithm that Counts. In *International Conference on Formal Methods in Computer-Aided Design*, pages 52–59. IEEE, 2012.

Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

COMPASS. COMPASS Project Web Page: `http://compass.informatik.rwth-aachen.de`, 2016. URL `http://compass.informatik.rwth-aachen.de`.

Matthew J Daigle, Xenofon D Koutsoukos, and Gautam Biswas. An Event-based Approach to Integrated Parametric and Discrete Fault Diagnosis in Hybrid Systems. *Transactions of the Institute of Measurement and Control*, 2009.

Rami Debouk, Stéphane Lafortune, and Demosthenis Teneketzis. On an Optimization Problem in Sensor Selection. *Discrete Event Dynamic Systems*, 12(4):417–445, 2002.

Maria D Di Benedetto, Stefano Di Gennaro, and Alessandro D'Innocenzo. Verification of Hybrid Automata Diagnosability by Abstraction. *IEEE Transactions on Automatic Control*, 56(9):2050–2061, 2011.

Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan. Fault-adaptivity in hard real-time component-based software systems. In *Software engineering for self-adaptive systems II*, pages 294–323. Springer, 2013.

ECSS-E-ST-70-11C. Space engineering; Space segment operability. Technical report, 2008.

ECSS-Q-ST-30-02C. Space product assurance; Failure modes, effects (and criticality) analysis (FMEA/FMECA). Technical report, 2009.

ECSS-Q-ST-40-12C. Space product assurance; Fault tree analysis – Adoption notice ECSS/IEC 61025. Technical report, 2008.

N. Eén and N. Sorensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

European Space Agency. Statement of Work: FDIR Development and Verification & Validation Process, 2011. Appendix to ESTEC ITT AO/1-6992/11/NL/JK.

FAME. FAME Project Web Page: `http://es.fbk.eu/projects/fame_main`, 2016. URL `http://es.fbk.eu/projects/fame_main`.

A. Feldman, T. Kurtoglu, S. Narasimhan, S. Poll, D. Garcia, Johan de Kleer, Lukas Kuhn, and A.J.C. van Gemund. Empirical Evaluation of Diagnostic Algorithm Performance Using a Generic Framework. *International Journal of Prognostics and Health Management*, Sep 2010. ISSN 2153-2648.

Marco Gario. *A Formal Foundation of FDI Design via Temporal Epistemic Logic*. PhD thesis, University of Trento, 3 2016. Fulltext available at https://marco.gario.org/phd/.

Alban Grastien. Symbolic Testing of Diagnosability. In *International Workshop on Principles of Diagnosis*, 2009.

Joseph Y Halpern and Judea Pearl. Causes and explanations: A structural-model approach. part i: Causes. *The British journal for the philosophy of science*, 56(4):843–887, 2005.

Joseph Y Halpern and Moshe Y Vardi. The complexity of Reasoning About Knowledge and Time. Lower Bounds. *Journal of Computer and System Sciences*, 38(1):195–237, 1989.

Thomas A Henzinger. The Theory of Hybrid Automata. In *Verification of Digital and Hybrid Systems*, pages 265–292. Springer, 2000.

Thierry Jéron, Hervé Marchand, Sophie Pinchinat, and Marie-Odile Cordier. Supervision Patterns in Discrete Event Systems Diagnosis. In *International Workshop on Discrete Event Systems*, pages 262–268. IEEE, 2006.

S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A Polynomial-time Algorithm for Diagnosability of Discrete Event Systems. *IEEE Transactions on Automatic Control*, 46(8):1318–1321, 2001.

Shengbing Jiang and R Kumar. Failure Diagnosis of Discrete Event Systems with Linear-Time Temporal Logic Fault Specifications. In *American Control Conference*, volume 1, pages 128–133. IEEE, 2002.

Shengbing Jiang, Ratnesh Kumar, and Humberto E Garcia. Diagnosis of Repeated/Intermittent Failures in Discrete Event Systems. *IEEE Transactions on Robotics and Automation*, 19(2):310–323, 2003a.

Shengbing Jiang, Ratnesh Kumar, and Humberto E Garcia. Optimal Sensor Selection for Discrete-event Systems with Partial Observation. *IEEE Transactions on Automatic Control*, 48(3):369–381, 2003b.

Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

François Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal Logic with Forgettable Past. In *Symposium on Logic in Computer Science*, pages 383–392, 2002.

Florian Leitner-Fischer and Stefan Leue. Causality checking for complex system models. In *Verification, Model Checking, and Abstract Interpretation*, pages 248–267. Springer, 2013.

Boyu Li, Ting Guo, Xingquan Zhu, and Zhanshan Li. Reverse Twin Plant for Efficient Diagnosability Testing and Optimizing. *Engineering Applications of Artificial Intelligence*, 38:131–137, 2015.

Agnes Madalinski, Farid Nouioua, and Philippe Dague. Diagnosability Verification with Petri Net Unfoldings. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 14(2):49–55, 2010.

Robin McDermott, Raymond J Mikulak, and Michael Beauregard. *The basics of FMEA*. SteinerBooks, 1996.

K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

K.L. McMillan. Interpolation and SAT-based Model-checking. *Lecture Notes in Computer Science*, pages 1–13, 2003.

Amit Misra. *Senor-based diagnosis of dynamical systems.* PhD thesis, Vanderbilt University, 1994.

Amit Misra, Janos Sztipanovits, Al Underbrink, Ray Carnes, and Byron Purves. Diagnosability of dynamical systems. In *Third International Workshop on Principles of Diagnosis*, 1992.

Christophe Morvan and Sophie Pinchinat. Diagnosability of Pushdown Systems. In *Haifa Verification Conference*, pages 21–33. Springer, 2009.

Sergio Mover, Alessandro Cimatti, Ashish Tiwari, and Stefano Tonetta. Time-aware Relational Abstractions for Hybrid Systems. In *International Conference on Embedded Software*, pages 1–10. IEEE, 2013.

Sriram Narasimhan and Gautam Biswas. Model-based diagnosis of hybrid systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 37(3):348–361, 2007.

Stanley C Ofsthun and Sherif Abdelwahed. Practical applications of timed failure propagation graphs for vehicle diagnosis. In *Autotestcon, 2007 IEEE*, pages 250–259. IEEE, 2007.

Stephen Oonk and Francisco J Maldonado. Automated maintenance path generation with bayesian networks, influence diagrams, and timed failure propagation graphs. In *IEEE AUTOTESTCON, 2016*, pages 1–9. IEEE, 2016.

Joël Ouaknine and James Worrell. Some recent results in metric temporal logic. In *Formal Modeling and Analysis of Timed Systems*, pages 1–13. Springer, 2008.

Vilfredo Pareto. *Manuale di economia politica*, volume 13. Società Editrice Libraria, 1906.

Ludovic Pintard, Christel Seguin, and Jean-Paul Blanquart. Which automata for which safety assessment step of satellite fdir? In *Computer Safety, Reliability, and Security*, pages 235–246. Springer, 2012.

Claudia Priesterjahn, Christian Heinzemann, and Wilhelm Schafer. From timed automata to timed failure propagation graphs. In *Object/Component/Service-Oriented Real-Time Distributed Computing*

(ISORC), 2013 IEEE 16th International Symposium on, pages 1–8. IEEE, 2013.

Raymond Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32:57–95, 1987.

Jussi Rintanen and Alban Grastien. Diagnosability testing with satisfiability algorithms. In *International Joint Conference on Artificial Intelligence*, 2007.

Ana Rugina, Cristiano Leorato, and Elena Tremolizzo. Advanced validation of overall spacecraft behaviour concept using a collaborative modelling and simulation approach. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on*, pages 262–267. IEEE, 2012.

Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. Diagnosability of Discrete-event Systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

Meera Sampath, Raja Sengupta, Stephane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. Failure Diagnosis using Discrete-event Models. *IEEE Transactions on Control Systems Technology*, 4(2): 105–124, 1996.

Meera Sampath, Stephane Lafortune, and Demosthenis Teneketzis. Active Diagnosis of Discrete-event Systems. *IEEE Transactions on Automatic Control*, 43(7):908–929, 1998.

Leonardo Santoro, Marcos Moreira, and Joao Basilio. Computation of Minimal Diagnosis Bases of Discrete-Event Systems: Method of the Trees of Event Sets. In *Anais do XX Congresso Brasileiro de Automatica*, 2014.

Anika Schumann and Yannick Pencolé. Scalable diagnosability checking of event-driven systems. In *International Joint Conference on Artificial Intelligence*, pages 575–580, 2007.

Shane Strasser and John Sheppard. Diagnostic alarm sequence maturation in timed failure propagation graphs. In *AUTOTESTCON, 2011 IEEE*, pages 158–165. IEEE, 2011.

Xingyu Su, Marina Zanella, and Alban Grastien. Diagnosability of Discrete-Event Systems with Uncertain Observations. In *International Joint Conference on Artificial Intelligence*, pages 1265–1271, 2016.

Stavros Tripakis. Fault Diagnosis for Timed Automata. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 205–221. Springer, 2002.

Dirk van Dalen. *Logic and structure (3. ed.).* Universitext. Springer, 1994.

W Vesely, F Goldberg, N Roberts, and D Haasl. Fault tree handbook (nureg-0492). *Washington, DC: Division of Systems and Reliability Research, Office of Nuclear Regulatory Research, US Nuclear Regulatory Commission*, 1981.

W. Wang, S. Lafortune, and F. Lin. Optimal Sensor Activation in Controlled Discrete Event Systems. In *IEEE Conference on Decision and Control*, pages 877–882. IEEE, 2008.

Brian C Williams and P Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 971–978, 1996.

Songyan Xu, Shengbing Jiang, and Ratnesh Kumar. Diagnosis of Dense-time Systems Under Event and Timing Masks. *IEEE Transactions on Automation Science and Engineering*, 7(4):870–878, 2010.

Lina Ye and Philippe Dague. An Optimized Algorithm for Diagnosability of Component-based Systems. pages 143–148, 2010.

Lina Ye, Philippe Dague, Delphine Longuet, Laura Brandán Briones, and Agnes Madalinski. Fault manifestability verification for discrete event systems. In *European Conference on Artificial Intelligence*, pages 1718–1719, 2016.

Tae-Sic Yoo and Stéphane Lafortune. NP-Completeness of Sensor Selection Problems Arising in Partially Observed Discrete Event Systems. *IEEE Transactions on Automatic Control*, 47(9):1495–1499, 2002a.

Tae-Sic Yoo and Stéphane Lafortune. Polynomial-time Verification of Diagnosability of Partially Observed Discrete Event Systems. *IEEE Transactions on Automatic Control*, 47(9):1491–1495, 2002b.

Appendix A

TFPG-to-SMV

With the translation rules presented in Section 5.1, we produce for the small running example the following SMV model. The corresponding state space is shown in Figure 5.2b.

```
MODULE main

  IVAR trans_type : {NODE_ACTIVATION, MODE_CHANGE, TIME_TICK};
  VAR system_mode : {M1,M2};

  VAR failuremode_Stuck : failuremode (trans_type);
  VAR or_node_Overheat : or_node_2 (edge_Stuck_to_Overheat_1,
      edge_Stuck_to_Overheat_2, trans_type);
  VAR edge_Stuck_to_Overheat_1 : edge (1, 2, FALSE, failuremode_Stuck,
      or_node_Overheat, trans_type, system_mode=M1);
  VAR edge_Stuck_to_Overheat_2 : edge (0, 1, FALSE, failuremode_Stuck,
      or_node_Overheat, trans_type, system_mode=M2);

  TRANS trans_type = MODE_CHANGE <-> system_mode != next(system_mode);
  TRANS trans_type = NODE_ACTIVATION <->
    failuremode_Stuck.status != next(failuremode_Stuck.status) |
    or_node_Overheat.status != next(or_node_Overheat.status);
  TRANS or_node_Overheat.must_fire -> trans_type != TIME_TICK;

MODULE failuremode (trans_type)
  VAR status : {OFF, ACTIVE};
  ASSIGN init(status) := {OFF, ACTIVE};
  ASSIGN next(status) := case
    trans_type != NODE_ACTIVATION : status;
    status = OFF : {OFF, ACTIVE};
    TRUE : status;
    esac;

MODULE or_node_2 (edge1, edge2, trans_type)

  DEFINE can_fire := (edge1.can_fire | edge2.can_fire) & !must_fire;
  DEFINE must_fire := edge1.must_fire | edge2.must_fire;

  VAR status : {OFF, ACTIVE};
  ASSIGN init(status) := OFF;
  ASSIGN next(status) := case
      trans_type != NODE_ACTIVATION : status;
      can_fire : {OFF, ACTIVE};
      must_fire : ACTIVE;
      TRUE : status;
      esac;

MODULE edge (tmin, tmax, tmax_is_infinity, source, target, trans_type,
             system_mode_is_compatible)

  DEFINE is_active := source.status = ACTIVE &
                      system_mode_is_compatible &
                      target.status = OFF;
  DEFINE can_fire := is_active & counter >= tmin &
      (counter < tmax | tmax_is_infinity);
  DEFINE must_fire := is_active & counter = tmax &
      !tmax_is_infinity;

  VAR counter : 0..tmax;
  ASSIGN init(counter) := 0;
  ASSIGN next(counter) := case
      next(!is_active) : 0;
      counter < tmax & trans_type=TIME_TICK : counter + 1;
      TRUE : counter;
      esac;
```