UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

A Formal Foundation of FDI Design via Temporal Epistemic Logic

Marco Elio Gustavo Gario

Advisor

Prof. Alessandro Cimatti

Fondazione Bruno Kessler

March 2016

Abstract

Autonomous systems must be able to detect and promptly react to faults. Fault Detection and Identification *components (FDI) are in charge of detecting the occurrence of faults. The FDI depends on the concrete design of the system, needs to take into account how faults might interact, and can only have a partial view of the run-time state through sensors. For these reasons, the development of the FDI and certification of its correctness and quality are difficult tasks. This difficulty is compounded by the fact that current approaches for verification of the FDI rely on manual inspection and testing.*

Motivated by industrial case-studies from the European Space Agency*, this thesis proposes a formal foundation for FDI design that covers specification, validation, verification, and synthesis. The* Alarm Specification Language *(ASL_K) is introduced in order to formally capture a set of interesting and desirable properties of the FDI components. ASL_K is equipped with a semantics based on* Temporal Epistemic Logic*, thus enabling reasoning about partially observable systems. Automated reasoning techniques can then be applied to perform validation, verification, and synthesis of the FDI. This formal process guarantees that the generated FDI satisfies the designer expectations.*

The problems deriving from this process were out of reach for existing model-checking techniques. Therefore, we develop novel and efficient techniques for model-checking temporal epistemic logic over infinite state systems.

Keywords

[Fault Detection and Identification, Temporal Epistemic Logic, Model Based Design, Model Checking, Diagnosis]

Acknowledgments

I would like to thank everybody in the ES unit at FBK, for the lighthearted fun, and for always making me feel welcome and at home. A special thanks goes to Alessandro Cimatti for guiding me through this journey and for the challenging discussions in front of the whiteboard. I had the honor and pleasure to work side by side with Marco Bozzano and Stefano Tonetta, and they both taught me a lot. I am especially grateful to Andrea, Cristian and Sergio, for their support in dealing with the many things that are bound to occur in such a complex endeavor.

My internship at NASA Ames was an awesome experience that would have not been possible without the support of Kristin Yvonne Rozier.

Thanks to Alicia and Mauro, for always pushing me to follow my interests, and fostering my curiosity; Carolina, for caring and leading the way; Tatiana, for being understanding of all the traveling and deadlines, and bringing joy to my life. Thank you for coping with the long silences during stressful periods, and support during times of doubt. Thanks also to all my geo-distributed family. I hope this thesis will answer the dreaded question: *what are you studying?*

Contents

List of Tables

List of Figures

Acronyms Table

ASL	Alarm Specification Language
BDD	Binary Decision Diagrams
BMC	Bounded Model Checking
BSS	Battery Sensor System
DES	Discrete Event System
FDIR	Fault Detection Identification and Recovery
FMECA	Failure Mode Effect and Criticality Analysis
FTA	Fault Tree Analysis
LTS	Labeled Transition System
MBD	Model Based Diagnosis
PUS	Packet Utilization System
SLIM	System-Level Integrated Modeling
SMT	Satisfiability Modulo Theory
TEL	Temporal Epistemic Logic
TFPG	Timed Failure Propagation Graph

Chapter 1

Introduction

Autonomous systems, such as satellites, space rovers, and self-driving cars, need to be able to react quickly to unexpected events such as faults. In the case of spacecrafts, this need is compounded by the difficulty (or impossibility) of directly accessing the system to repair it. A wide range of techniques have been developed to address this issue, among which Fault Detection Identification and Recovery (FDIR) is a common example in the aerospace setting. The goal of the FDIR is to autonomously detect (FDI) and recover (FR) from faults of the system in a timely manner.

The design of FDIR in aerospace is a very complex task, as it needs to consider the interaction between heterogeneous components, implemented either in hardware or in software, and take into account their interactions also in presence of multiple faults. Additionally, the interaction of these components might be regulated by mission- or safety-critical requirements that also need to be taken into account.

By definition, the FDIR comes into play during critical phases of the system life, and might be the last resort for preserving the spacecraft. For this reason, it is paramount to verify the correctness and completeness of the FDIR. This need, however, clashes with the current industrial practices, where FDIR development is mostly considered as an additional activity

rather than a core architectural concept. This causes several problems, such as a lack of standardization and dedicated tools. In turn, this leads to limited reuse of existing designs, ad-hoc handling of special cases, and overly complex or overly simplified FDIR designs. Since changes to details of the system design can have drastic impacts on the FDIR strategy, the FDIR is currently developed late in the system development life-cycle. These facts make the certification of the FDIR quality a daunting task and, in some cases, they can lead to increases in cost and delays in the spacecraft development.

In different areas of engineering, there has been a shift towards *model-based* techniques [130]. By having an artifact capture explicit characteristic of the system, it is possible to highlight key aspects in the design and avoid ambiguity when collaborating with other people. Moreover, models can be equipped with semantics and used for automated analysis such as simulation.

Formal methods are a family of techniques that can help reasoning about complex situations by providing a formal, i.e., mathematically grounded, characterization of the problem. In particular, *model checking* is an active area of research in which the model of the system is equipped with a formal semantics and properties can be verified against it. Model checking provides an exhaustive analysis of the system behavior. Therefore, if the property is shown to be satisfied by the model, it means that there is no possible execution of the model that can violate it. This is different from other techniques such as testing, where not finding a counter-example does not prove that no counter-example exists. An important assumption of model-based techniques is that the model is a faithful representation of the system. This makes it extremely important to be able to validate the model of the system, before performing other analysis. If the model is a formal model, we can apply several automated reasoning techniques to

increase our confidence in the model.

In this thesis, we propose the use of formal model-based techniques to support early FDI design and verification. Motivated by ESA-funded industrial projects, we develop the theory and tools to support the specification, validation, verification, design and synthesis of FDI. In order to properly capture the specification of FDI requirements, we show that logics that can reason about partial-observability are needed. For this reason, our approach is built on top of Temporal Epistemic Logic (TEL). TEL is commonly used in the context of multi-agent systems, in order to reason about interacting agents. We show that this logic is particular suitable for reasoning about the problem of the FDI. Moreover, we show how many important information (such as which sensors can be observed, and how many observations can be used for reasoning) can be embedded in the semantics of the logic, thus obtaining a unified logical characterization of the problem. In order to tackle the complex model-checking problems arising from the application of our approach, we develop novel and efficient techniques for model-checking TEL over infinite state systems.

1.1 Contributions

To improve the FDIR design process, we need to define a way to specify, validate and verify both our requirements, and all the models used in the process. Moreover, the goal of improving formal model-based FDIR design cannot be achieved without effective tools to perform automated reasoning. For this reason, the contributions of this thesis span across two domains: FDI design and temporal epistemic logic model-checking. The following are the contributions of the thesis.

1. Formally characterize several key aspects of FDI, and consolidate the specification by introducing the Alarm Specification Language

(ASL_K). ASL_K is equipped with a semantics based on temporal epistemic logic, that accounts for different types of recall and enables automated reasoning (e.g., model-checking)

2. Develop algorithms for the synthesis of FDI components that satisfy an ASL_K specification by construction; [34]

3. Extend a classical approach for diagnosability (i.e., the twin-plant approach) in order to deal with different types of ASL_K specifications.

4. Develop an algorithm for optimizing the selection of sensors to be used by the FDI according to multiple cost-function (Pareto optimal sensor placement); [23]

5. Introduce techniques for the validation of Timed Failure Propagation Graphs based on Satisfiability Modulo Theory (SMT) engines; [31]

6. Describe the application of the techniques presented in this thesis, within the two ESA-funded projects AUTOGEF and FAME. [25]

7. Develop the first approach for model-checking of a particular type of TEL (KL_1 with observational semantics) over infinite state transition systems, that is also extremely effective on finite state systems. [49]

1.2 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 provides some technical background that is shared across multiple Chapters. Chapter 3 provides an overview of the state of the art in both the fields of diagnosis, FDI design and temporal epistemic logic reasoning.

Chapters 4 to 8 develop the main technical part concerning FDI design, including: specification, validation, verification, and synthesis. In particular:

- Chapter 4 describes the setting of this work, e.g., what the plant and the FDIR are, and how they are connected. Moreover, it introduces the running example of the Battery Sensor System.
- Chapter 5 formally defines several key concepts for the specification of the FDI, such as completeness, correctness, maximality and trace diagnosability.
- Chapter 6 introduces the Alarm Specification Language (ASL_K), its temporal epistemic grounding, and examples of its application.
- Chapter 7 discusses the twin-plant approach to diagnosability, how to apply it to ASL_K specifications, and how to exploit it to optimize the amount of sensors in a system.
- Chapter 8 presents two algorithms for the synthesis of FDI components. These algorithms are characterized by the amount of recall of the FDI.

Chapter 9 discusses the use of Timed Failure Propagation Graphs to model the propagation of failures in a system, and discusses SMT-based techniques for their validation.

Chapter 10 describes the application of the techniques described in this thesis within two ESA-funded projects: AUTOGEF and FAME.

The techniques for performing TEL model-checking are introduced in Chapter 11. In particular, we introduce two algorithms (eager and lazy) for model-checking TEL over both finite and infinite state systems. The effectiveness of the approaches is demonstrated through an extensive experimental evaluation.

Chapter 12 concludes the thesis by providing an outlook on future work.

Chapter 2

Technical Background

We use the standard notions of first-order logic for theory, satisfiability, validity, unsatisfiability, and logical consequence [106]. We refer to 0-arity predicates as *Boolean variables*, and to 0-arity uninterpreted functions as *theory variables*. Given a finite set of variables V and a (potentially infinite) domain $\mathcal{U}$ of values, an assignment to V is a mapping from the set V to the set $\mathcal{U}$. We use $\Sigma(V)$ to denote the set of assignments to V. Given an assignment $s \in \Sigma(V)$ and $V_1 \subseteq V$, we use $s_{|V_1}$ to denote the *projection* of s over V_1. We use $\mathcal{F}(V)$ to denote the set of propositional formulas over V. If $V_1, \ldots, V_n$ are a sets of variables and φ is a formula, we might write $\varphi(V_1, \ldots, V_n)$ to indicate that all the variables occurring in φ are elements of $\bigcup_{i=1}^{n} V_i$. Given a theory $\mathcal{T}$, we write $\varphi \models_{\mathcal{T}} \psi$ (or simply $\varphi \models \psi$) to denote that the formula ψ is a logical consequence of φ in the theory $\mathcal{T}$.

2.1 Satisfiability Modulo Theory

Given a first-order formula ψ with non-logical symbols interpreted in a decidable background theory $\mathcal{T}$, *Satisfiability Modulo Theory* (SMT) [12] is the problem of deciding whether there exists a satisfying assignment to the free variables in ψ. SMT extends the Satisfiability problem (SAT) with theories, e.g., arithmetic theories. This provides a simple and intu-

itive way to encode several types of problems, that require more than just propositional logic, for example, formal verification [20] and temporal reasoning [54]. The existence of effective solvers such as MathSAT [53], Z3 [66] and CVC4 [11] (to name a few) make SMT a practical approach. An example of the theories that we use is the theory of linear arithmetic over the rational numbers ($\mathcal{LRA}$). A formula in $\mathcal{LRA}$ is an arbitrary Boolean combination, a universal ($\forall$) or an existential ($\exists$) quantification, of atoms in the form $\sum_i a_i x_i \bowtie c$ where $\bowtie \in \{>, <, \leq, \geq, \neq, =\}$, each x_i is a real variable, each a_i and c are real constants.

2.2 Labeled Transition System

To model reactive systems (such as the plant and the FDIR), we use a symbolic representation of *Labeled Transition Systems* (LTS). Control locations and data are represented by variables, sets of states and transitions are represented by formulas, and transitions are labeled with explicit events. For each variable x, we assume that there exists a corresponding variable x' (the *primed version* of x). If V is a set of variables, V' is the set obtained by replacing each element x with its primed version ($V' = \{x' \mid x \in V\}$). Given an assignment s to variables in V, we denote with s' the assignment to the variables V' such that $s'(v') = s(v)$ for every $v \in V$. Given a formula φ, φ' is the formula obtained by adding a prime to each variable occurring in φ.

Definition 1 (LTS)**.** *A* Labeled Transition System *is a tuple* $S = \langle V, E, I, T \rangle$*, where:*

- V *is a finite set of state variables;*
- E *is a set of events;*
- $I \in \mathcal{F}(V)$ *is a quantifier-free formula over* V *defining the initial states;*

- $T : E \to \mathcal{F}(V \cup V')$ *maps an event* $e \in E$ *to a quantifier-free formula over* V *and* V' *defining the transition relation for* e *(with* V' *being the next version of the state variables).*

A *state* s is an assignment to the state variables V (i.e., $s \in \Sigma(V)$). We denote by s' the corresponding assignment to V'. A transition labeled with e is a pair of states $\langle s, s' \rangle$ such that $\langle s, s' \rangle \models T(e)$. A [finite] *trace* of S is an infinite [resp. finite] sequence $\sigma = s_0, e_0, s_1, e_1, s_2, \ldots$ alternating states and event such that s_0 satisfies I and, for each $k \geq 0$, $\langle s_k, s_{k+1} \rangle$ satisfies $T(e_k)$. Note that we consider infinite traces only, and w.l.o.g. we assume the system to be dead-lock free, and call $\Sigma(S)$ the set of all traces of S. Given $\sigma = s_0, e_0, s_1, e_1, s_2, \ldots$ and an integer $k \geq 0$, we denote by σ^k the finite prefix $s_0, e_0, \ldots, s_k$ of σ containing the first $k+1$ states. We denote by $\sigma[k]$ the $k+1$-th state s_k. A state s is *reachable* in S iff there exists a trace σ of S such that $s = \sigma[k]$ for some $k \geq 0$.

Definition 2 (Deterministic LTS)**.** *S is* deterministic *iff:*

(i) there is one initial state (i.e., there exists a state s such that $s \models I$ and, for all t, if $t \models I$, then $s = t$);

(ii) for every reachable state s, for every event e, there is one successor (i.e., there exists s' such that $\langle s, s' \rangle \models T(e)$ and, for all t', if $\langle s, t' \rangle \models T(e)$, then $s' = t'$).

We say that S is a *finite transition system* if the set of events E is a finite set, and all domains of the variables in V are finite; otherwise, we say that S is an *infinite transition system*. The infinite characterization of these systems is given by the infinite domain of the variables, i.e., in each state we have a finite number of variables that can potentially have an infinite domain. For example, we can have integer or rational values, and use the theory of arithmetic [73] to define the transition relation.

Definition 3 (Synchronous Product)**.** *Let*

$$S^1 = \langle V^1, E^1, I^1, T^1 \rangle \text{ and } S^2 = \langle V^2, E^2, I^2, T^2 \rangle$$

be two transition systems with $E^1 = E^2 = E$ *and* $V^1 \cap V^2 = \emptyset$*. We define the* synchronous product $S^1 \times S^2$ *as the transition system* $\langle V^1 \cup V^2, E, I^1 \wedge I^2, T \rangle$ *where, for every* $e \in E$, $T(e) = T^1(e) \wedge T^2(e)$*. Every state* s *of* $S^1 \times S^2$ *can be considered as the product* $s^1 \times s^2$ *such that* $s^1 = s_{|V^1}$ *is a state of* S^1 *and* $s^2 = s_{|V^2}$ *is a state of* S^2*. Similarly, every trace* σ *of* $S^1 \times S^2$ *can be considered as the product* $\sigma_1 \times \sigma_2$ *where* σ_1 *is a trace of* S^1 *and* σ_2 *is a trace of* S^2*.*

Definition 4 (Asynchronous Product)**.** *Let*

$$S^1 = \langle V^1, E^1, I^1, T^1 \rangle \text{ and } S^2 = \langle V^2, E^2, I^2, T^2 \rangle$$

be two transition systems, such that $V^1 \cap V^2 = \emptyset$*. We define the* asynchronous product $S^1 \otimes S^2$ *as the transition system* $\langle V^1 \cup V^2, E^1 \cup E^2, I^1 \wedge I^2, T \rangle$ *where:*

- *for every* $e \in E^1 \setminus E^2$, $T(e) = T^1(e) \wedge \mathit{frame}(V^2)$*.*
- *for every* $e \in E^2 \setminus E^1$, $T(e) = T^2(e) \wedge \mathit{frame}(V^1)$*.*
- *for every* $e \in E^1 \cap E^2$, $T(e) = T^1(e) \wedge T^2(e)$*.*

where $\mathit{frame}(X)$ *stands for* $\bigwedge_{x \in X} x' = x$ *and is used to represent the fact that while one transition system moves on a local event, the other transition system does not change its local state variables.*

Every state s of $S^1 \otimes S^2$ can be considered as the product $s^1 \otimes s^2$ such that $s^1 = s_{|V^1}$ is a state of S^1 and $s^2 = s_{|V^2}$ is a state of S^2. If either S^1 or S^2 is deterministic, then every trace σ of $S^1 \otimes S^2$ can be considered as the product $\sigma_1 \otimes \sigma_2$ where σ_1 is a trace of S^1 and σ_2 is a trace of S^2. In general, the product of two traces produces a set of traces due to

different possible interleaving of the local events. In general, composing two systems can reduce the behaviors of each system and introduce deadlocks. For example, one system can reach a state where it can only perform the event e, but the other system is never ready to synchronize on e. However, given two systems (e.g., the diagnoser and the plant), if one of the systems is deterministic (the diagnoser) then it cannot alter the behavior of the second (the plant).

Notice that the synchronous product coincides with the asynchronous case when the two sets of events coincide. Sometimes we are interested in the synchronous behavior of systems, without the need of considering multiple possible events. If the set of events is a singleton (e.g., $E = \{tick\}$), we call the system a *Transition System* (TS), and omit the set of events: $S = \langle V, I, T \rangle$. The general definition of LTS is more general, and can capture both types of systems.

2.3 Linear Temporal Logic

A formula in Linear Temporal Logic extended with past operators [133, 112, 114] (or simply LTL), is defined over variables V and events E as:

$$\begin{aligned} \beta ::= & \; p \mid e \mid \beta \wedge \beta \mid \neg\beta \mid \\ & O\beta \mid H\beta \mid Y\beta \mid \beta S \beta \mid \\ & F\beta \mid G\beta \mid X\beta \mid \beta U \beta \end{aligned}$$

where p is a predicate over $\mathcal{F}(V)$ and $e \in E$. Intuitively, p are the propositions over the state of the LTS, while e represents an event.

Given a trace $\sigma = s_0, e_0, s_1, e_1, s_2, \ldots$, the semantics of LTL is defined as follows:

- $(\sigma, n) \models p$ iff $s_n \models p$

- $(\sigma, n) \models e$ iff $e_n = e$
- $(\sigma, n) \models \beta_1 \wedge \beta_2$ iff $(\sigma, n) \models \beta_1$ and $(\sigma, n) \models \beta_2$
- $(\sigma, n) \models \neg\beta$ iff $(\sigma, n) \not\models \beta$
- Once: $(\sigma, n) \models O\beta$ iff $\exists j \leq n.\ (\sigma, j) \models \beta$
- Historically: $(\sigma, n) \models H\beta$ iff $\forall j \leq n.\ (\sigma, j) \models \beta$
- Yesterday: $(\sigma, n) \models Y\beta$ iff $n > 0$ and $(\sigma, n-1) \models \beta$
- Since: $(\sigma, n) \models \beta_1 S \beta_2$ iff there exists $j \leq n$ such that $(\sigma, j) \models \beta_2$ and for all k, $j < k \leq i$, $(\sigma, k) \models \beta_1$
- Finally: $(\sigma, n) \models F\beta$ iff $\exists j \geq n.\ (\sigma, j) \models \beta$
- Globally: $(\sigma, n) \models G\beta$ iff $\forall j \geq n.\ (\sigma, j) \models \beta$
- Next: $(\sigma, n) \models X\beta$ iff $(\sigma, n+1) \models \beta$
- Until: $(\sigma, n) \models \beta_1 U \beta_2$ iff there exists $j \geq n$ such that $(\sigma, j) \models \beta_2$ and for all k, $n \leq k < j$, $(\sigma, k) \models \beta_1$.

Given an LTS S, and an LTL formula β, S *satisfies* β ($S \models \beta$) iff for every trace σ of S, $(\sigma, 0) \models \beta$.

Notice that $Y\beta$ is always false in the initial state, and that we use a reflexive semantics for the operators U, F, G, S and O. We use the following abbreviations:

- $X^n\beta \stackrel{\Delta}{=} XX^{n-1}\beta$ (with $X^0\beta = \beta$);
- $Y^n\beta \stackrel{\Delta}{=} YY^{n-1}\beta$ (with $Y^0\beta = \beta$);
- $O^{\leq n}\beta \stackrel{\Delta}{=} \beta \vee Y\beta \vee \cdots \vee Y^n\beta$;
- $F^{\leq n}\beta \stackrel{\Delta}{=} \beta \vee X\beta \vee \cdots \vee X^n\beta$.

2.4 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. In this thesis, we consider the linear time extension KL [93] that combines the epistemic operator K with LTL (including past operators). A KL formula has the following syntax:

$$\begin{aligned}\beta ::= & \ p \mid e \mid \beta \wedge \beta \mid \neg\beta \mid \\ & O\beta \mid H\beta \mid Y\beta \mid \beta S\beta \mid \\ & F\beta \mid G\beta \mid X\beta \mid \beta U\beta \mid \\ & K_i\beta\end{aligned}$$

where i is one of the (finitely many) agents that can observe the system (e.g., diagnosers). The intuitive semantics of $K_i\beta$ is that the agent *knows* that β holds in a state of a trace σ, by using only the observable information. Each modal operator K_i is associated with an *indistinguishability relation* $\sim_i \subseteq (\Sigma(S) \times \mathbb{N}) \times (\Sigma(S) \times \mathbb{N})$ that defines when two pointed traces are indistinguishable from each other. Intuitively, given two traces σ_A, σ_B and two points on them n_A, n_B, we say that we cannot distinguish (σ_A, n_A) from (σ_A, n_B) iff $((\sigma_A, n_A), (\sigma_A, n_B)) \in \sim_i$. This means that $K_i\beta$ holds iff β holds in all situations that are indistinguishable (e.g., observationally equivalent). This abstraction allows us to reason about the knowledge of an agent that has limited information on the system execution. In Section 5.3, we will explain how to build this relation to account for the type of recall of the agent.

While in LTL the interpretation of a formula is local to a single trace, in KL the semantics of the K_i operator quantifies over the set of indistinguishable traces. The semantics of KL is defined recursively on pointed traces and indistinguishability relations, and it mostly extends the seman-

tics of LTL. Given a trace $\sigma = s_0, e_0, s_1, e_1, s_2, \ldots$ of S, and $n \geq 0$ we have that:

- $(S, \sigma, n) \models p$ iff $s_n \models p$
- $(S, \sigma, n) \models e$ iff $e_i = e$
- $(S, \sigma, n) \models \beta_1 \wedge \beta_2$ iff $(S, \sigma, i) \models \beta_1$ and $(S, \sigma, n) \models \beta_2$
- $(S, \sigma, n) \models \neg\beta$ iff $(S, \sigma, n) \not\models \beta$
- $(S, \sigma, n) \models O\beta$ iff $\exists j \leq n.\ (S, \sigma, j) \models \beta$
- $(S, \sigma, n) \models H\beta$ iff $\forall j \leq n.\ (S, \sigma, j) \models \beta$
- $(S, \sigma, n) \models Y\beta$ iff $n > 0$ and $(S, \sigma, n-1) \models \beta$
- $(S, \sigma, n) \models \beta_1 S \beta_2$ iff there exists $j \leq i$ such that $(S, \sigma, j) \models \beta_2$ and for all k, $j < k \leq i$, $(S, \sigma, k) \models \beta_1$
- $(S, \sigma, n) \models F\beta$ iff $\exists j \geq n.\ (S, \sigma, j) \models \beta$
- $(S, \sigma, n) \models G\beta$ iff $\forall j \geq n.\ (S, \sigma, j) \models \beta$
- $(S, \sigma, n) \models X\beta$ iff $\sigma, n+1 \models \beta$
- $(S, \sigma, n) \models \beta_1 U \beta_2$ iff there exists $j \geq n$ such that $(S, \sigma, j) \models \beta_2$ and for all k, $i \leq k < j$, $\sigma, k \models \beta_1$.
- $(S, \sigma, n) \models K_i\beta$ iff for all traces σ' of S and integers $m \geq 0$ s.t. $(\sigma, n) \sim_i (\sigma', m)$ it holds that $(S, \sigma', m) \models \beta$.

To keep the notation lighter (and more in line with the one used for LTL) we usually omit the system S and write (σ, n) instead of (S, σ, n) when the system is clear from the context. A system S satisfies a formula β ($S \models \beta$) if for every trace σ of S $(S, \sigma, 0) \models \beta$.

From KL we derive several syntactic fragments. In particular, we call KL_n the fragment of KL in which at most n nesting of epistemic operators K_i occur. Using this definition, we can see that KL_0 is LTL. The restriction applies only to nesting, while we consider the possibility of multiple agents interacting. In this thesis, we are mostly interested in KL_1 since, as we will show in Chapter 6, it allows us to encode all interesting properties of alarms.

2.5 Parameter Synthesis

In model-checking we are interested in knowing if a given property is satisfied in a given system. In some cases, we are interested in knowing whether a family of systems satisfy a property. In particular, we are interested in knowing what changes we can make to a system in order to satisfy the property. To achieve this, we introduce parameters that guide the behavior of the system. We then ask the model-checker to provide us with all values for the parameters such that the property is satisfied.

Let $S = \langle V, E, I, T \rangle$, be an LTS, and U a set of parameters. The *parametric LTS* Q is defined as $Q = (V, E, U, I_U, T_U)$, where $I \in \mathcal{F}(U \cup V)$ is the initial condition, and $T : E \to \mathcal{F}(U \cup V \cup V')$ is the transition relation. Parameters do not change over time, and can assume any domain that a variable can assume. Parameters have an impact on both which states can be considered initial, and on which transitions are possible within the system. In fact, given a valuation for the parameters ($\gamma \in \Sigma(U)$), and a formula ψ we denote by $\gamma(\psi)$ the formula obtained by substituting in ψ every occurrence of u with $\gamma(u)$ for every parameter u in U. Given a parametric transition system Q and a valuation for the parameters γ, we can compute the *induced* LTS by replacing the parameters with their valuation: $Q_\gamma = (V, E, \gamma(I_U), \gamma(T_U))$.

Definition 5 (Parameter Synthesis Problem)**.** *Given a parametric system Q and a property φ, the* parameter synthesis problem *is the problem of finding the set of all valuations* $\textsc{ValidPars}_{Q,\varphi}$ *s.t. the induced LTS satisfies φ: i.e.,*

$$\textsc{ValidPars}_{Q,\varphi}(U) \triangleq \{\gamma \in \Sigma(U) \mid Q_\gamma \models \varphi\}$$

Definition 5 is independent from the type of the property φ, and we will consider both invariant properties (i.e., propositional formulas that must hold in any state of a trace) and LTL properties. Moreover, we do not make assumptions on Q, and thus we work with both to finite state and to infinite state systems. Since model-checking for infinite state systems is undecidable, the parameter synthesis problem is also undecidable. In practice, however, we are usually able to solve it when the number of parameters is limited. In this work, we rely on off-the-shelf tools for LTL model-checking and parameter synthesis for infinite-state systems [47].

Chapter 3

State of the Art

This thesis tries to bridge two different areas of research: FDI design, and temporal epistemic logic. Both domains have been obect of extensive research. In order to clarify the positioning of this thesis, this chapter provides an overview of both fields.

In Section 3.1, we discuss the relation between diagnosis and FDI, and we overview several key aspects and works related to diagnosis of reactive systems. This leads us to the discussion of challenges and issues related to FDI design.

In Section 3.2, we provide an overview of the techniques and results obtained in the field of temporal epistemic logic model-checking.

3.1 Diagnosis and FDI

FDI design is strongly connected to the field of diagnosis. In fact, in both areas we are interested in understanding why a system is misbehaving. Over the last 30 years, the field of diagnosis has expanded in order to address different challenges and assumptions. The goal of this section is to clarify the relation between FDI and diagnosis, and thus the position of this thesis within the state of the art.

There are at least three main approaches for performing diagnosis (that

are not necessarily mutually exclusive): rule-based, data-driven, and model-based.

Rule-based approaches provide a simplified description of the diagnoser behavior that hard codes the relation between observations and causes. This information is encoded within rules that are manually developed by systems experts. These techniques come from the tradition of expert systems [155]. A rule is expressed as a condition and effect. Sets of rules constitute a knowledge-base. The observations coming from the system are fed to the knowledge base to see whether they imply the occurrence of a fault either directly or through a chain of rules.

Rule-based systems are hard to maintain, since the relation between observable behavior of the system and faults is captured explicitly, (typically) without the use of a model of the system. This requires a deep knowledge of the system, and it is not robust w.r.t. the introduction of new faults or changes in the sensors sets. Nevertheless, these systems are particularly useful in those contexts where external knowledge is available (e.g., human knowledge) that we want to capture and formalize in the diagnostic engine. This can be done without the need of modeling the whole system, that might be complex or partially unknown (e.g., biological systems).

Data-driven approaches rely on available historical data in order to characterize and learn nominal behaviors from off-nominal ones. Examples include pattern recognition [125] and machine learning techniques [119].

These techniques work without a model, or with just a simplified one. This makes it possible to easily adjust them to changes in the system, or even adapt them to different systems. The main drawback of these techniques is the need of data for training. Data from the systems is not available until they have been built. Only during the testing phase of the physical system, it is possible to collect useful data. This, however, drastically shortens the time available for the tuning of the diagnoser, and

makes it extremely difficult to validate the diagnoser when designing the system.

In *Model-based* approaches, the model captures the behavior of the system. Whenever the observations are not consistent with the model, we try to find a justification. In this thesis, we focus on model-based techniques. The reasons are many-fold. First, in different engineering areas, the use of models has become prominent, in order to better capture and formalize the expectations and assumptions on the system. Indeed, Model Based Engineering is becoming a more widely accepted practice, and it is even dictated by standards [130] for the development of avionics systems. Second, there is a significant amount of literature on model-based diagnosis. Finally, by using a formal model of the system we can apply formal methods techniques to deduce and prove properties of the system. It is worth noticing, however, that not all model-based techniques require a formal model to work. UML diagrams are a typical example of models without an official formal semantics.

There are two main limitations to model-based approaches. First, the system might be complex, the design might change, or it might be difficult to get details on the implementation of certain parts of the system. This represents a difficulty in generating the actual model, and keep it up-to-date during the iterations of the system design. This point will be mitigated by the availability of models in the model-based engineering process, and by the development of effective translations. Second, modeling big systems usually requires performing some sort of abstraction in order to keep the model reasonable in size. Choosing the right level of abstraction is often difficult. Using the wrong level of abstraction can lead to systems that cannot be analyzed or that are too simple.

3.1.1 Model-Based Diagnosis

The Model-Based Diagnosis (MBD) approach was introduced by Reiter [134]. In MBD the system model is defined as description of the components behavior. Observations are run against the model, and if they are inconsistent, it means that some component is behaving incorrectly. The goal of MBD is to find the set of constraints that are in conflict with the observation. In this way, a *diagnosis* is a set of components that must misbehave in order to justify the observation. The number of diagnosis suffers from a combinatorial explosion. Therefore, Reiter also proposes an algorithm to perform minimal diagnosis. A diagnosis is minimal if any subset of it is not a diagnosis. Upon this idea, several approaches and techniques have been developed, in order to improve the performances both at the algorithmic and implementation level.

To categorized the different model-based techniques, we consider the following aspects:

- Temporal Evolution
- Faults Modeling
- Output of the Diagnoser
- Resources available to the Diagnoser

Temporal Evolution MBD originated in the *combinational* domain of circuit design. Extensions of MBD to sequential system have been proposed, and [45] provides an overview of the different aspects considered in the literature. In particular, the sequential nature of the problem has an impact on the behavior of faults that are not only permanent anymore but can have transient dynamics (i.e., appear and disappear).

The most common framework for extending MBD to *sequential* system is the one developed by Sampath [137] on top of Discrete Event Systems (DES). This approach has been widely adopted, helped by the fact that DES are commonly used outside of the diagnosis community.

Other techniques to deal with timed behavior include chronicles [113], Timed Failure Propagation Graphs [4], and timed automata [151]. In all these cases, the time is not considered discrete, but continuous.

Faults Modeling Faults effects are difficult to capture within models. Things can break in many unexpected ways. Therefore, in the literature we find two possible ways to model faults: strong and weak model [64]. A *strong* model describes exactly how the fault affects the system. For example, a *stuck at closed* fault in a valve, means that the valve will stay closed no matter what commands is given to it. Weak fault models, instead, originate from the abductive reasoning domain, and simply indicate that a component is violating one of its constraints [146, 139, 38]. Weak faults models are simpler to introduce, at the expense of less informative diagnosis.

Another aspect to consider in fault modeling is their dynamics and occurrence. The dynamics tells us whether a fault is permanent or transient. For transient faults, there might be additional constraints on how long they can last, and how often they can repeat. In most cases, we consider the occurrence of a fault as a non-deterministic event. Nevertheless, in safety critical domains, information concerning the reliability of the components is usually available. Therefore, we might want to attach a probability to the occurrence of a fault. This is considered in many MBD approaches, in which the probabilities are used to prioritize the diagnosis. An example of probabilistic fault modeling is provided within the COMPASS [36] toolset.

Output of the Diagnosis An important aspect of the FDIR module is that its complexity is limited by the number of available reconfiguration actions [17]. FDIR is not meant to perform deep state inspection, as suggested by MBD approaches, since this is left to the ground control (using telemetry data). Thus, the FDIR module only checks whether reconfiguration actions should be applied. For FDI we are interested in providing enough information to the FR in order to apply the correct recovery action. This provides us with a more restrictive set of outputs for the FDI component, that we call *alarms*. For this reason, the FDI and the Diagnoser output different information.

The classical definition of diagnosis in the MBD approaches [134] is one (or more) sets of components whose malfunction can justify the discrepancy in the observation. The richer the explanation of the problem, the fastest the problem can be identified and solved [10].

Since the diagnoser provides a rich set of information, it becomes harder to judge the quality of the diagnoser. In fact, multiple empirical metrics have been defined [83, 82] to assess the quality of the diagnoser. This contrasts our goal of formally certifying the quality of the FDI at design time.

We will focus on answers that are qualitative: either the fault has occurred, or it has not. However, this black and white view, might not always be desirable. For this reason, several works try to adopt a quantitative view of the problem, by assessing the likelihood of a diagnosis to be correct. This is typical, for example, of Bayesian techniques [142].

Resources for Diagnosis If the diagnoser does not have timing or computational constraints, then it can use more complex algorithms to compute a diagnosis. This is usually the situation during maintenance, or when the ground stations use telemetry. Since ground has more computational

power, it can run multiple complex algorithms at the same time. However, this introduces delays in the reaction to faults and depends also on the possibility of communicating with the spacecraft. In the context of FDI, instead, a short reaction time might make the difference between saving or losing the spacecraft.

One approach to avoid complex algorithms on-board, is to perform part of the reasoning at design time, and use *pre-compiled* information on-board. Compilation of diagnosis has been studied by using both BDDs [150, 140] and Decomposable Negation Normal Form (DNNF) [97]. These compilation approaches are exact, in the sense that all relevant information is considered, and the resulting artifact (BDD or DNNF) is correct by construction. The drawback, however, is that the compiled artifact is considerably big, and cannot be generated for models of significant size.

In order to limit the size of the monitors, in industrial use-cases, the type of monitors is simplified, by only using thresholds of values and simple counters. There is no assurance on the quality of the monitors, and they need to be separately validated and verified. However, in this approach, the complexity of the monitors is independent from the complexity of the overall system, and thus can be applied in complex designs.

Finally, the diagnoser might be able to use another type of resource: probes. The usual distinction is between *passive* and *active* diagnosis. Passive diagnosis consists in performing the diagnosis of the system without affecting its behavior. This is in contrast with active diagnosis, in which probes can be used to stimulate parts of the system [65].

Passive diagnosis has more limited information to act upon and therefore, with the same observation, it might generate many more diagnosis than an active diagnoser. Intuitively, the active diagnoser can remove candidate diagnosis and reduce ambiguity by probing the system. Depending on the framework being considered, performing an observation on the sys-

tem might have a cost [84] or be a potentially disruptive action, since it impacts the nominal operation of the system [48] (e.g., turn on/off a component). For this reason, most approaches for autonomous systems are based on passive techniques.

3.1.2 DES Diagnosis

Diagnosis of Discrete Event System finds its roots in the work by Sampath et al. [137]. In that paper, the authors propose an approach for testing the diagnosability of DES, and synthesizing the diagnoser through a subset construction. By doing so, they obtain a diagnoser that is able to state whether a fault occurred or not. Several extensions to this work consider extensions and relaxed some assumptions of this seminal work.

First of all, the diagnosability test proposed by Sampath et al. requires to build the diagnoser. Since this is an expensive step, [101] proposes a polynomial time[1] technique for testing the diagnosability of the DES: the twin-plant approach. In this way, the construction of the diagnoser can be performed only for those systems that are diagnosable.

Due to the exponential construction, the size of the DES remains the limiting factor in the applicability of these techniques. Several works try to tackle the problem, by showing how to break the model of a global diagnoser into multiple local diagnosers [141, 10, 102].

The original approach is limited to permanent faults. However, in practice, we are interested in explaining more complex behaviors. In [111] and [100] supervision patterns are used to provide richer explanations of what the fault means for the system.

Most approaches assume that all observations are available from the beginning, a notable exception is [148], in which techniques to consider windows of observations are proposed. Moreover, they discuss whether by

[1]Using an explicit, i.e., not symbolic, representation of the DES

Technique	System Type	Time Model	Sensing	Output	Runtime
Model-Based	**Sequential**	**Discrete**	**Passive**	**Alarms**	**Compilation**
Data-Driven	Combinational	Continuous	Active	Diagnosis	On-the-Fly
Rule-Based				Probabilities	

Figure 3.1: Positioning of this work

recording an additional bit of information, it is possible for the diagnoser to improve its accuracy. Observations are crucial for performing diagnosis. In certain contexts, however, assuming that all observations are perfect might be a strong assumption. In [110], different types of uncertainty in the observations are considered, including uncertainty between the order of observations, their provenance, or their concrete values.

Continuous time behavior (as opposed to discrete time) is considered in [151]. By limiting the analysis to non-zeno path, it is possible to extend the concept of delay to the timed context. Authors also provide an algorithm to construct the explicit diagnoser using Difference Bound Matrices [18].

Probabilities to the possibility of being in a given state are considered in [149], by using Stochastic DES. An algorithm is provided to build a diagnoser that, at every transition, updates the probability of the system of being in a given state. Therefore, given a sequence of observations, it is possible to compute the probability of being in a faulty condition, and thus attach a likelihood to each diagnosis.

3.1.3 FDIR Design

Autonomy on spacecraft is a long standing objective [126]. The goal of the FDIR is to make the system react to unexpected events, and preserve the safety and mission of the spacecraft.

In this thesis, we focus on *model-based* techniques for *discrete-time reactive systems*. An FDI is a system that performs *passive diagnosis* that

has an *alarm as output* in order to trigger a recovery. In order to be able to verify the FDI, and to use it on-board autonomous systems, we are interested in *compilation approaches*. Figure 3.1 provides a schematic overview of the positioning of this work with respect to the landscape of the state of the art. The different dimensions that need to be considered are mostly unrelated to each other. We consider the combination highlighted in bold, due to its relevance in the FDIR setting.

To design the FDIR we need data concerning the hardware dependability [86], such as Failure Mode Effect Criticality Analysis (FMECA [156]), Fault Tree Analysis (FTA [156]), Common Cause Failure Analysis, and Hazard Analysis. Unfortunately, this data becomes available late in the life-cycle when development has already started. This impacts the FDIR specification [75] and development, since the quality of FDIR depends on data that is not available during early design phases. In turn, this leads to a low FDIR maturity, or can introduce delays in the project. No dedicated approach to FDIR development exists, which can be employed starting form the early system development phases, and which is able to take into account the design from both, Software and System (including Hardware) perspective [75]. The existing approaches are specific (both in terminology and application) to each company [132].

The FDIR needs to consider all combinations of faults and nominal behaviors [17], and might need to account for embedded FDIR capabilities of sub-components. Therefore, the design of FDIR components is a challenging task by itself. Due to its complexity and importance, there is a need of supporting the Verification and Validation (V&V) process of the FDIR design [120].

In this setting, the ESA project COMPASS [36], represents an example application of formal model based techniques for the design and validation of spacecraft designs. In particular, COMPASS provides a language to

model the Software and System of the spacecraft, and run model-checking verification queries. In particular, it is possible to test the diagnosability of certain faults and, if FDIR components are provided within the model, it is possible to verify whether they can achieve detection and recovery. COMPASS provides a good starting point, however, it does not help in the process of specifying, nor designing the FDIR component.

The examples discussed so far are not unique of the European industry. Indeed, NASA has also identified the need for better Fault Management procedures. To achieve this goal, a handbook of Fault Management [128] has been under development since 2008. The goal of this manual is to collect the experience from multiple NASA centers, and industrial partners, and agree on a common terminology and strategy.

The ECSS standards define a way to perform on-board monitoring for aerospace devices. In particular, the Packet Utilization Standard (PUS) (Section 5.8.6 of [85]) *On-board Monitoring Service* can be configured to performed on-board monitoring. Those monitors consider [129]:

- An exact value (up to a bit mask)
- A value within a lower-/upper-threshold
- A delta-monitoring in which the last values of the change in the value of the parameter should be within a threshold

Each monitor supports a repetition value whose semantics depends on the nature of the check to be performed. For limit-check and expected values, this is the number of successive samples of the parameter that can (or must) satisfy a condition before establishing a new status for the parameter. For a delta-check, this is the number of consecutive delta values to be used to evaluate an average delta value which is checked against the delta check definition (i.e., average over successive pairs of values).

All monitors can then be conditioned to a validity check, this is an expression over other parameters that indicates whether the monitor should be considered or not (e.g., a GPS unit providing off-nominal readings is ignored if the GPS is off).

The use of PUS On-board monitoring systems simplifies (and limits) the type of FDI components that can be implemented. Monitoring exact values and lower-/upper-threshold requires the assumption that the current value of the sensors is sufficient to detect and identify the faults. Indeed, we do not need to recall the observations, but only whether they violated the condition. For delta-checks, instead we need to keep track of the values in order to be able to compute the average over the last repetitions. In any case, we are considering a limited amount of observations and memory. This is in contrast with the typical DES diagnosis view [137] in which we assume that all observable events are considered (i.e., perfect-recall).

Verification Formal techniques have been applied to the verification of FDIR in a few works.

COMPASS [36] is a project (and set of tools) for the formal development of aerospace systems. Among others, it provides a way to perform diagnosability analysis. Moreover, the verification tools can be applied to check the behavior of an FDIR component. In [17] the authors explore a few alternatives for formal modeling and verification of FDIR sub-components, using tools such as OCAS [21, 37], and SCADE [70]. In [29], timing constraints in the FDIR recovery process are verified using the UPPAAL [16] model-checker.

All these works focus on the application of formal techniques for the verification of the FDIR design. The model of the system is formally captured, but the specification of the FDIR behavior is not. Therefore, the properties used to verify the FDIR are defined by the designer on a case-by-case

bases. In order to achieve autonomy, however, verification of components against ad-hoc properties is not sufficient. Instead of specifying *how* the FDIR should behave, we need to be able to specify *what* we want to achieve and *what* can be done within the system [126].

3.2 Model Checking Temporal Epistemic Logic

Model-checking is a field of formal verification that is starting to be successfully applied in industry. Using model-checking tools, a user can verify a given property against a model of the system [8]. The main benefits of model-checking are the exhaustive search of the behavior of the system, the production of a counter-example (in case the property is violated), and the use of a model of the system that can be reused or derived from other artifacts of the design process.

In this thesis, we focus on *symbolic model-checking* [121]. In symbolic model-checking, the set of states is represented as a symbolic expression, i.e., a formula. In this way, it is possible to describe large (or even infinite) sets using a finite representation. Symbolic verification on finite models is a consolidated field. Verification of infinite models (e.g., timed, hybrid) is a newer field. Verifying infinite state models even for simple properties is (in general) undecidable. Nevertheless, from the practical standpoint, there have been many advancements related to both the identification of decidable fragments, and the development of incomplete algorithms.

3.2.1 Specification Languages

The properties that we want to verify on the model can be expressed in different ways. Among the most common property specification types we have *reachability*, linear time logic (LTL) and computation tree logic (CTL).

Reachability is concerned with whether a given state (usually considered

a "bad" state) can be reached from an initial configuration. The dual problem (invariant checking) is concerned with verifying that all reachable states satisfy a certain (usually good) condition.

Reachability concerns states in isolation. In many case we are interested in the temporal evolution of the system. Temporal logics, such as LTL and CTL are used for this. The difference between the two logics is that the first considers each execution of the system in isolation. The second, instead, considers the tree-like structure generated by multiple traces, when a branching point is encountered. This makes it possible to consider multiple behaviors of the system in parallel. For example, in LTL is not possible to express a property concerning both branches of a conditional statement in a program.

Verification of these logics is interested in the actual behavior of the system. However, in situations like FDI design, we are interested in reasoning about the potential knowledge of an external observer. Temporal epistemic logics (TEL) are used to capture this type of properties. These logics are considerably used in the domain of multi-agent systems, in which multiple components (e.g., processes, autonomous systems) coexist and only have limited information about the actions of the other components. This requires verifying properties of relative knowledge. An example application is the *Bit Transmission Protocol*, in which two processes try to exchange information over an unreliable channel. The specification that we want to verify in this type of situation is not only that the message will be eventually correctly received, but that the sender (by using the limited observable information available to him) can *know* that the message was received correctly. Other interesting applications of TEL come from the the domain of information security [9], or cryptographic protocols [30], where we are interested in guaranteeing that some information will remain private, even if some public information is shared.

There are several ways of combining epistemic operators and temporal operators, giving raise to several logics. The extension of LTL is called sometimes KL [93] (sometimes LTLK [122]) and is the main focus on this thesis. The extension of CTL is called $CTLK$ [69, 94].

Due to the nature of multi-agent systems, it is interesting to study the problem of coalitions, i.e., whether a set of agents is able to achieve a goal by collaborating. $ATEL$ [99] is an example of such a logic.

Two key aspects come into play when extending a temporal logic with epistemic operators: which operators are included, and the recall of the agents. In particular, many extensions that include the common knowledge operator C_G (everybody knows that everybody knows that ...) are undecidable [93, 117]. The recall type is usually split in two types: perfect and bounded recall. In perfect recall, the agent can remember all observations from the beginning of time, while in bounded recall, only a fixed amount of observations are recalled (usually zero, sometimes called observational semantics). The type of recall also plays a significant role in the complexity of the reasoning algorithms.

CTLK The standard approach to CTL model-checking is based on Binary Decision Diagrams (BDDs [46]). The idea behind the approach is to navigate the syntax-tree of the formula, and recursively build the sets of states that satisfy the given subformula. This algorithm allows a simple extension to deal with $CTLK$ if we are considering the zero recall for the agents [116].

Apart from the BDD-based approach, a few works try to use SAT-based techniques to reason about $CTLK$. An example is [103] in which an unbounded model-checking technique is proposed.

To deal with perfect-recall, [69] proposes to use an oracle, in order to decide in which states the epistemic atoms are satisfied. This oracle

is obtained using a subset construction, in order to consider all possible states that are observationally equivalent for the agent.

The use of an oracle to decide the satisfaction of the epistemic atoms is further leveraged in [152]. In [152] the temporal epistemic model checking problem is reduced to temporal model checking, by manually introducing (expressions on) variables local to an agent that are satisfied if and only if the corresponding epistemic expression is satisfied.

LTLK Bounded model-checking [22] (BMC) is a popular (incomplete) technique for model-checking LTL properties using a SAT-based engine. Intuitively, this technique works by defining constraints on the trace that we want to find, and asking the SAT solver to find such a path.

Extensions of this approach to $LTLK$ under observational semantics are limited to the positive fragment, in which the epistemic operator K cannot appear negated within the formula [122, 157]. The reason being that with BMC we can only reason about one trace at the time while, in general, the epistemic operator requires us to reason on multiple paths. This problem is overcome in [153] (for perfect recall) by building an oracle for the epistemic atoms, using a subset construction (similar to [69]). However, this approach still requires the computation of the reachable states, and thus might not be able to completely leverage the performance of SAT-based technologies.

IC3 BDDs are not able to deal with industrial size designs that can easily overcome 10^{200} states. For this reason, we are interested in applying SAT/SMT based algorithms. IC3 [42] is a recent SAT-based algorithm for the verification of invariant properties on finite state systems. This technique has shown impressive performances in the Hardware model-checking competition, and is becoming one of the most used algorithms in the model-checking community. Extensions to IC3 include dealing with infinite-state

systems [51], and LTL properties [52] and CTL properties [95]. No work, however, has attempted to use IC3 to perform verification of temporal epistemic logic. In this thesis, we address this short-coming.

3.2.2 Models and Tools

Models in this domain are usually characterized by multiple agents (or processes) performing actions following a *protocol* (i.e., their programming). The outcome of these actions is regulated by an environment.

Most work on temporal epistemic model checking focuses on finite state models. Notable exceptions are theoretical works considering continuous-time [157], and infinite state systems (i.e., Artifact Centric Systems [15, 13]). Most algorithms for infinite state model-checking work by abstracting the problem into a finite representation. This either requires a restriction on the possible starting models [13], or an incomplete result [115].

However, tools for temporal epistemic logic model-checking are limited to the verification of finite state systems. In this thesis, we work towards addressing this limitation. In particular, we are interested in extending tools for model-checking temporal logics over infinite state transition systems (such as nuXmv [47]) in order to deal with epistemic modalities.

The state of the art model-checkers for TEL are are MCMAS [116] and MCK [88]. The two have a slightly different focus. MCMAS support only the logics $ATEL$ and $CTLK$ under observational semantics, and the reasoning engine is only based on BDDs. The limited scope is, however, balanced by an efficient implementation, that is able to deal with models of reasonable size (within the limits of the BDD technology).

MCK, instead, supports a wide range of logics, semantics and algorithms [68]. For example, it supports observational, clock and perfect recall semantics. Moreover, it has specialized algorithms for fragments of $CTLK$ and $LTLK$.

Chapter 4

Plant and FDIR

In our general setting, a plant is connected to components for Fault Detection and Identification (FDI), and for Fault Isolation and Recovery (FR), as depicted in Figure 4.1. The role of the FDI is to collect and analyze the observable information from the plant, and to turn on suitable alarms associated with (typically unobservable) conditions. The FR component receives the alarms from the FDI and applies suitable reconfiguration actions to mitigate and recover from the detected (and potentially harmful) situation.

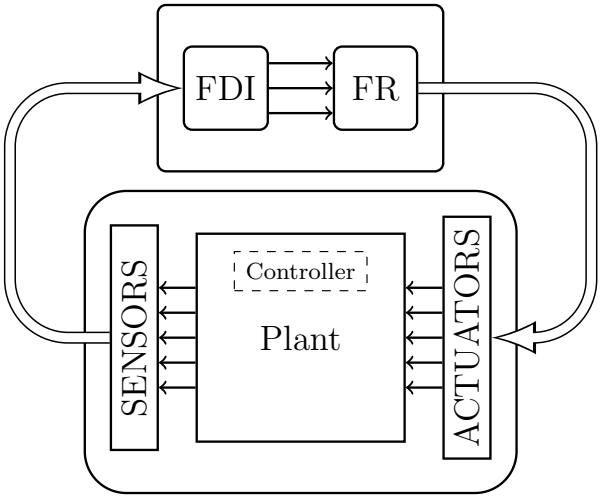

Figure 4.1: Integration of the FDIR and Plant

In this chapter, we informally describe the plant (Section 4.1), the FDIR

(Section 4.2), and how the two components are connected (Section 4.3). To connect the two components, we need to understand and define several concepts such as the modeling formalism of the plant, observability requirements, memory of observations, and synchronization. A formal characterization of all those concepts is given in Section 4.4. The Battery Sensor System, our running example, is presented in Section 4.5.

The contribution of this chapter is to clarify the relation between the FDIR components (FDI, FR, and FDIRConf) and between the plant and FDIR. This helps us better characterize the setting of the thesis. The formal definitions and the running example are extended versions of the ones presented in [34].

4.1 The Plant

The *plant* is the system that we want to diagnose and control. This can be anything ranging from an industrial plant, to a satellite or a space rover. In general, the plant is equipped with a nominal controller, that takes care of the nominal operation of the system. The nominal controller might have access to more sensors and actuators than the ones provided to the FDIR.

To properly capture the behavior of the plant, we need an expressive formalism. In this work, we consider both plants modeled as finite state systems, as well as plants modeled as infinite state systems, in order to better capture the dynamics of the underlying physical system. There is a clear trade-off between expressiveness of the modeling formalism and effectiveness of the reasoning tasks. For example, on finite state systems we can enumerate all possible states in which a system can be, thus guaranteeing termination for many reasoning tasks. This is not the case for infinite state systems, in which many reasoning tasks are undecidable. Another important distinction is whether we consider continuous or discrete

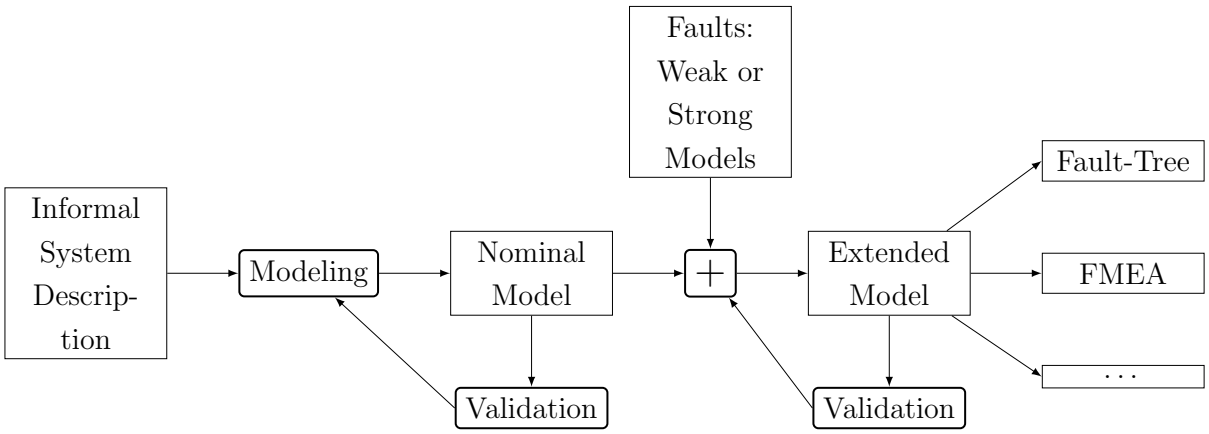

Figure 4.2: Plant Modeling Process

time dynamics. Literature on diagnosis from the AI and DX community has a stronger focus on finite state, discrete time systems [65, 134]. On the other hand control theory and FDI communities tends to work with infinite state, continuous time systems [127, 63].

We consider both finite and infinite state systems with discrete time dynamics. The only exception to this is Chapter 9 where we discuss Timed Failure Propagation Graphs, and the possibility of abstracting them from continuous time to discrete time.

As depicted in Figure 4.2, the first step in modeling the plant for FDIR design is to model it under *nominal* conditions, i.e., without faults. This should include the nominal controller, in order to be able to verify that the system fulfills the requirements in absence of faults. The case-studies and examples discussed in this thesis, have been modeled using the nuXmv [47] or the SLIM [39] language. Both languages can be used to model infinite state discrete time systems and can be model checked against specifications using, respectively, nuXmv and COMPASS [36]. This phase usually requires multiple iterations, in which the plant is checked against validation properties, until all behaviors of interest have been captured.

Once the model of the plant is completed, we can introduce fault mod-

els, by a process called *model fault extension*. In this step we define which faults are possible in the system and how they can affect it. We distinguish between two type of fault models: strong and weak. A *strong* fault model clearly defines the impact of the fault on the system. A typical example is a valve being *stuck at closed*. The precise characterization of the fault can give raise to more interesting diagnosis. However, defining fault models in such detail is a difficult task that requires a deep knowledge and understanding of the system. A *weak* fault model, instead, simply specifies that a component will not behave as expected, without defining how it will behave. This type of fault model is more general, and it is particularly suitable during the early stages of design, where details on the components might not be available. The use of weak versus strong fault model should be balanced against the ability for a diagnoser to perform accurately. A weak fault model allows the component to mimic the nominal behavior even in presence of a fault. This leads to faults that are not diagnosable (by construction) in the classical sense, and the need for finer definitions such as *trace diagnosability* (Section 5.5).

Once the model has been extended with faults, we can apply a wide range of techniques to validate it. These techniques come from the domain of model-based safety assessment and include Fault Trees [156], and Failure Mode Effects Analysis [136]. Moreover, we need to make sure that each fault can freely occur in the model. This can be done using techniques from receptiveness analysis [2].

4.2 The FDIR

The techniques presented in this work can be applied to a variety of systems. However, we are mainly interested in systems that are *autonomous*, with *limited computational power*, and limited to no access for manual

maintenance. A typical system matching this criteria is a *satellite* or a *space rover*. These systems pose constraints on space, energy and computation. Therefore, we focus on a *compilation approach*, in which the FDIR is designed offline, and compiled into an executable form that can be efficiently run on-board. This provides us with the need and opportunity of formally verifying the compiled FDIR, in order to formally certify that it meets our expectations.

The FDIR is divided into two sub-components: FDI and FR. These components are formally defined by providing a characterization of their expected behavior.

FDI The *FDI* takes sensors reading in input and estimates the state of the plant. In many approaches [134], the diagnoser outputs a set of faults that might have occurred. In our framework, we focus on the FDI ability of raising *alarms*. Intuitively, alarms are Boolean outputs associated with the occurrence of some situation of interest. This provides a clear Input/Output characterization of the FDI component: an FDI is a function that takes sensor readings in input and provides alarms as outputs (Figure 4.3). Alarms are commonly associated to fault detection, fault isolation, and fault identification. However, there is no reason to limit the scope of the alarms to faults, especially considering that the same fault in different situations might have a different severity, and thus need to be addressed in different ways.

One important aspects of the FDI is its ability to store and use historical observations. Intuitively, an FDI that can remember a longer window of observations might be able to perform more accurate diagnosis. The capability of storing previous observations is called *recall*. The classic definition of diagnosability for Discrete Event Systems [137], assumes *perfect recall*: the ability to recall all observations from the beginning of time. This

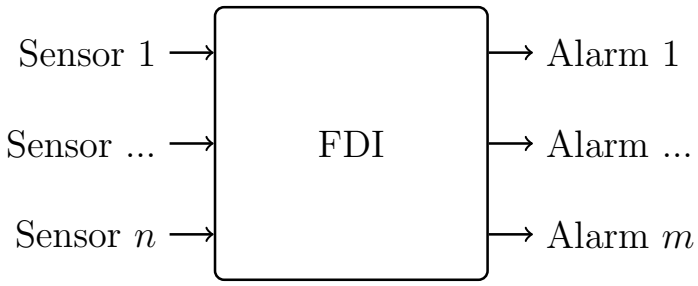

Figure 4.3: Abstract view of the FDI

yields the best possible diagnoser, however, it poses non trivial problems. For finite-state plants, the size of the memory to store those observations is exponential in the number of states of the plant. Even worse, for infinite-state plants, it is in general impossible to achieve perfect recall with finite memory. These issues motivate the idea of relaxing the *perfect recall* assumption and considering the *bounded recall* case, where a fixed-length window of observations is used. While this could seem like a simplistic assumption, this matches real-world usage and design of FDI, where most of the existing diagnosers use a limited amount of recall to decide whether the system experienced some fault. This is partially motivated by the simplicity of constructing this type of FDI, as a circuit taking in input the stored observations. Finally, focusing on a finite window of observations provides the ability to *forget* off-nominal observations: transient off-nominal sensor readings will be discarded after a certain amount of time, and thus will not impact the diagnoser forever; moreover, it is possible to *restart* the diagnoser, since we do not need all observations in order to guarantee the correct behavior of the diagnoser.

In this work we consider both perfect recall, due to its theoretical relevance, and bounded recall, due to its practical relevance.

FR The *FR* takes the alarms in input and performs recovery actions. The alarms work as triggers to start some predefined recovery sequence. The

objective of the recovery is to take the system into some desired (i.e., goal) configuration. The FR is a function that takes alarms in input and outputs a recovery plan. As an example, let us consider a power-supply subsystem. The FDI will raise an alarm if the primary power-supply component fails. This alarm will be used by the FR to start the recovery, which goal is to make the power-supply subsystem functional again by, for example, switching to a secondary power-supply component.

An in-depth discussion of FR design is out of the scope of this work. Techniques for FR design can be found in the AI literature on planning. As an example, in the AUTOGEF and FAME projects (see Chapter 10) we applied (respectively) techniques from *planning under partial observability* [19], and *conformant planning* [144] literature. In *planning under partial observability* the FR has access to some sensors of the system. The sensor data can be used to better reason on the non-determinism of the system. In this context, a plan is a sequence of actions with conditional branches. The conditions on the branches are expressed on the observable part of the system that the FR uses to decide how to continue the plan execution (e.g., if the light is green execute action A, otherwise action B). In *conformant planning* the FR does not have any access to the sensors of the system. A conformant plan needs to work independently of the non-determinism of the system. In this context, a plan is a sequence of actions, without conditional branching. Conformant plans are more difficult to find but they are easier to implement.

FDIRConf Separating the FDI from FR makes it possible to better deal with the specification of the requirements and the design of the solution: both components can be designed independently and in parallel. In this way, improvements to the FDI capabilities do not require the re-design of the FR (and vice-versa). This is particularly useful for validation and cer-

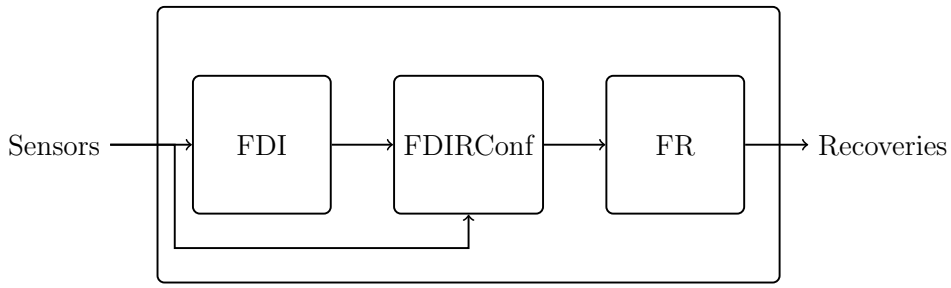

Figure 4.4: FDIRConf integration in the FDI

tification, as well as for experimentation of novel solutions. However, this poses the problem of boot-strapping the FDI, i.e., provide the possibility to the FDI to reason about the system during and after the FR execution. In our case-studies (Chapter 10), we address this problem by only allowing the FR to issue commands that are part of the nominal behavior of the system. In other words, any execution that includes FR activities exists as a nominal execution of the system. This makes the nominal model less diagnosable (in general), and a better integration of the two systems should be considered in the future.

This division of concerns requires *coordination*. For this reason, we introduce an additional component called *FDIRConf*. The FDIRConf is responsible for forwarding the alarms from the FDI to the FR. This provides a clean interface for dealing with the issue of *mission phases* and *operational modes*. The operational life of a system is composed by a succession of phases that have different characteristics, these are usually called *mission phases*. Launch, insertion to orbit, and science are examples of the different mission phases of a satellite. Moreover, there might be different operational conditions for the spacecraft. These are called *operational modes* and usually indicate which parts of the system are enabled. Each phase and mode might have different FDIR requirements: i.e., different alarms and different recovery goals. We use the FDIRConf to manage this complexity. In particular, we design the FDI in order to always provide all

the alarms, and let the FDIRConf suppress the ones that are not relevant for the current phase and mode. Active alarms are then routed to the recovery trigger that is relevant for the current phase and mode. Figure 4.4 provides a more detailed picture of the FDIR component, when including the FDIRConf. The FDIRConf can use information coming from the sensors in order to understand which are the current mission phase and operational mode. This type of design is used in our case studies and further discussed in Chapter 10.

4.3 Connecting the Plant and the FDIR

We described the the Plant and the FDIR separately, and now we focus on the key issues to consider when connecting these two components together. The FDIR interacts with the plant through *actuators* and *sensors*. We consider the actuators to be simple events, and focus on the relation between sensors and FDI accuracy. This requires us to discuss:

1. observability and domain of observations,
2. synchronicity of the observations,
3. and sampling of the observations.

Observability We assume sensors to only be able to access the observable part of the system. The plant might model physical quantities, that have an infinite domain characterization, e.g., real-valued domains. Therefore, we allow sensor readings to have both finite and infinite domain. In many practical situations, however, the exact value of an infinite domain sensor reading is not important. What matters, instead, is the historical trend of the value [44], whether it is below/above a certain threshold, or whether it matches a specific value within an error range. From a sensor that

provides an infinite domain reading (e.g., a real-valued tachometer) we can derive multiple finite domain sensors, that provide us an abstracted view of the quantity that we are measuring. Therefore, it is not the case that all infinite state models require infinite domain sensors.

Given a plant and some requirement for the FDI, the only two ways to improve the accuracy of the FDI are to increase the number of sensors, or to increase the recall of the FDI. Therefore, the set of sensors in the model plays a crucial role on the effectiveness of the FDI. We will formalize this concept when discussing diagnosability in Chapter 7.

Synchronous vs Asynchronous Once we have defined which observations are shared among the plant and FDIR, we need to define *when* they are shared. This boils down to defining whether the two systems evolve *synchronously* or *asynchronously*. In the synchronous case, observations and alarms are constantly synchronized between the FDI and the plant. This means that no observation can go undetected by the FDI. However, this also means that the FDI is not able to perform a diagnosis in between observations. In the asynchronous setting, instead, observations and alarms are updated only during synchronization points (i.e., events). Thus, observations that occur in-between two synchronization points might not be seen by the FDI. On the other hand, this allows both systems to evolve independently, and at different speed.

In [33] we use synchronous composition to connect the plant to the FDIR. The synchronous case can be seen as a particular case of the asynchronous case, by considering every event as a synchronization event. Therefore, as done in [34], in this work we use the more general setting of asynchronous composition.

Discrete vs Continuous Time In the synchronous composition, the FDI and plant evolve at the same speed. However, in the asynchronous composition, this is the case only if both systems have a continuous time dynamic, but not if both have a discrete time dynamic. This is particularly interesting, because the FDI might be able to perform diagnosis in-between two synchronizations, by keeping track of how much time has passed. We will discuss this behavior in Chapter 9 when presenting the validation of Timed Failure Propagation Graphs, that is a formalism that is inherently continuous time. Nevertheless, in Chapter 9 we will also show how we can go from a continuous to a discrete time plant, and how to design the FDI accordingly.

4.4 Formal Definitions

A *partially observable LTS* is an LTS $S = \langle V, E, I, \mathcal{T} \rangle$ extended with a set $E_o \subseteq E$ of observable events. Observations on state variables are used in practice, however, they make the formalism less clear. We limit the formalism only to observations on events and, whenever observations on state variables are needed, we incorporate them in the events (as done in [138]). Notice that extending the events set with observations over infinite domain variables will lead to an infinite set of events.

A *plant* $P = \langle V^P, E^P, I^P, T^P, E_O^P \rangle$ is a partially observable labeled transition system. An FDI component (or diagnoser) is a machine D that synchronizes with observable traces of the plant P. D has a set $\mathcal{A}$ of alarms that are activated in response to the monitoring of P. We use the general model of asynchronous composition to combine the diagnoser with the plant through observable events.

Definition 6 (Diagnoser)**.** *Given a set $\mathcal{A}$ of alarms and a partially observable plant $P = \langle V^P, E^P, I^P, T^P, E_O^P \rangle$, a diagnoser is a deterministic*

LTS $D(\mathcal{A}, P) = \langle V^D, E^D, I^D, T^D \rangle$ *such that* $E^D \subseteq E_o^P$, $V^P \cap V^D = \emptyset$ *and* $\mathcal{A} \subseteq V^D$.

When clear from the context, we use D to indicate $D(\mathcal{A}, P)$.

We assume that the plant and diagnoser are composed asynchronously: i.e., $D \otimes P$. Only observable events are used to perform synchronization. All the events of the diagnoser are observable events of the plant. This means that the diagnoser does not have internal transitions: every transition of the diagnoser is associated with an observable transition of the plant. This means that the diagnoser is a deterministic LTS. Having a deterministic diagnoser is useful because it makes it easier to understand how it will react to the observations coming from the plant:

Definition 7 (Diagnoser Matching trace)**.** *Given a diagnoser* D *of* P *and a trace* σ_P *of* P*, the diagnoser trace matching* σ_P*, denoted by* $D(\sigma_P)$*, is the trace* σ *of* D *such that* $\sigma \otimes \sigma_P$ *is a trace of* $D \otimes P$.

The notion of *diagnoser matching trace* is well defined because D is deterministic, and therefore there exists one and only one trace in D matching σ_P.

In [34] we required the set of events of the diagnoser to coincide with the set of observable events of the plant ($E^D = E_o^P$). In this work, we relax this condition, in order to be able to study the behavior of diagnosers that have access to different sets of observables. For example, a diagnoser D_1 might have access to more sensors than another diagnoser D_2 ($E^{D_1} \supset E^{D_2}$), or to a different set of sensors ($E^{D_1} \cap E^{D_2} = \emptyset$). This allows us to reason about relative knowledge when having access to different amount of information. This idea is at the base of the sensor placement problem, as discussed in Chapter 7.

Definition 8 (Observable Trace)**.** *Let* E_o *be the set of observable events of the partially observable transition system* P*, and let* σ *be a trace of* P.

The observable trace *associated with prefix* σ^k *of* σ *is defined recursively as follows:*

- $obs_{E_o}(\sigma^0) = \epsilon$ *(empty sequence);*

- *if* $e \in E_o$*, then* $obs_{E_o}(\sigma^k, e, s) = obs_{E_o}(\sigma^k), e$*;*

- *if* $e \notin E_o$*, then* $obs_{E_o}(\sigma^k, e, s) = obs_{E_o}(\sigma^k)$*.*

Since we are in an asynchronous setting, we allow the diagnoser to update its knowledge of the observables during particular points: i.e., during synchronization events.

Definition 9 (Observation Point)**.** *We say that* i *is an* observation point *for* σ*, denoted by* $ObsPoint(\sigma, i)$*, iff the last event of* σ^i *is observable, i.e., iff* $\sigma^i = \sigma', e, s$ *for some* σ', e, s *and* $e \in E_o$*.*

The notion of two traces being observationally equivalent requires that the two traces end both (or neither) in an observation point. This captures the idea that a trace ending in an observation point can be distinguished from the same trace extended with local unobservable steps. In other terms, an observer can distinguish the instant in which it is observing and an instant right after.

In many situations, we are interested in considering formulas only at observation points. We do so by introducing the following abbreviation:

Definition 10 (Observed)**.** *If* E_o *is the set of observable events, given a formula* ϕ*, we use* $\llcorner\phi\lrcorner$ *(read "Observed* ϕ*") as abbreviation for* $\phi \wedge Y \bigvee_{e \in E_o} e$*.*

This notation is useful to stress that something can happen only during synchronization events. We say that the alarm A is *triggered* when A is true after the diagnoser synchronized with the plant (i.e., when $\llcorner A\lrcorner$ is true).

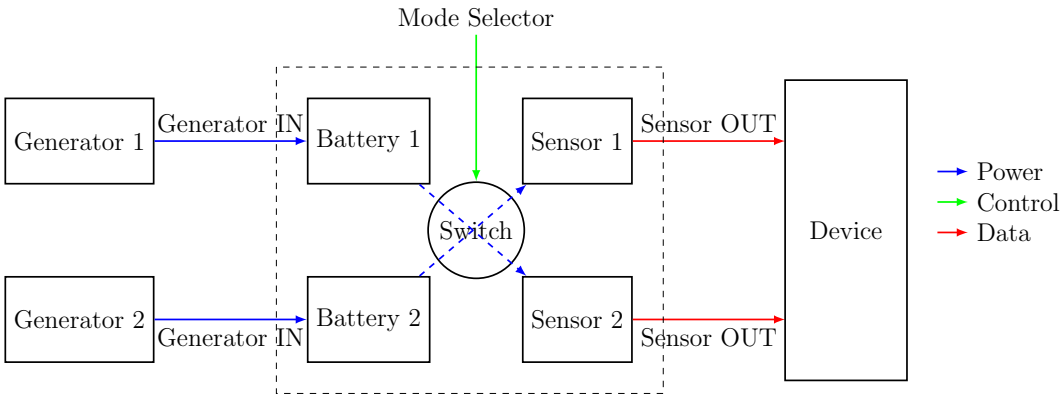

Figure 4.5: Battery Sensor System

4.5 The Battery Sensor System Example

The *Battery Sensor System* (BSS) (Figure 4.5) will be our running example [34]. The BSS provides a redundant reading of the sensors to a device. Internal batteries provide backup in case of failure of the external power supply. The safety of the system depends on both of the sensors providing a correct reading. The system can work in three different operational modes: *Primary*, *Secondary 1* and *Secondary 2*. In Primary mode, each sensor is powered by the corresponding battery. In the Secondary modes, instead, both sensors are powered by the same battery; e.g., during Secondary 1, both Sensor 1 and Sensor 2 are powered by Battery 1. The Secondary modes are used to keep the system operational in case of faults. However, in the secondary modes, the battery in use will discharge faster.

We consider two possible recovery actions: i) Switch Mode, or ii) Replace the Battery-Sensor Block (the dotted block in Figure 4.5). In order to decide which recovery to apply, we are going to define a set of requirements connecting the faults to alarms. The faults and observable information of the system are shown in Figure 4.6.

This example is particularly interesting because we can define two sources of delay: the batteries, and the device resilience to wrong inputs.

Observables	**Possible Values**
Mode	Primary, Secondary 1, Secondary 2
Battery Level {1, 2}	High, Mid, Low
Sensors Delta	Zero, Non-Zero ($\|S1.Out - S2.Out\| = 0$)
Device Status	On, Off

Component	**Faults**
Generator	Off ($G1_{Off}, G2_{Off}$)
Battery	Leak ($B1_{Leak}, B2_{Leak}$)
Sensor	Wrong Output ($S1_{WO}, S2_{WO}$)

Figure 4.6: Observables and Faults Summary

The batteries provide a buffer for supplying power to the sensors. The size of this buffer is determined by the capacity of the battery, the initial charge, and the discharge rate. For the device, we assume that two valid sensor readings are required for optimal behavior, however, we can work in degraded mode with only one valid reading for a limited amount of time. The device will stop working if both sensors are providing invalid readings, or if one sensor has been providing an invalid reading for too long.

Both a synchronous and asynchronous version of this model are possible. In the asynchronous model, we have an event for each possible combination of observations (e.g., "Mode Primary & Battery 1 Low"). In the synchronous model, we also have an additional observable event (*tick*) that represents the passing of time in the absence of any observable event. This event forces the synchronization of the plant with the diagnoser. The key difference between the synchronous and asynchronous setting is the amount of information that we can infer in this particular case. For example, if we know the initial charge level of a battery, and we know its discharge rate (given by the operational mode), then at each point in time we can infer the current charge of the battery. By comparing our expectation with the available information, we can detect when something is not behaving as expected. Unfortunately, there are practical settings in which the assumption of synchronicity is not realistic. Therefore, our approach accounts for both the synchronous and asynchronous models.

To provide a better understanding of how the running example behaves, we provide the LTS of each of the components. Figure 4.7 shows the LTS

of the generator and switch. We assume that the only way the generator can turn off is if a fault event occurs, thus the model of the generator is rather simple. Also the switch features a rather simple model, where the labels *toS1* and *toS2* are defined as:

- toS1: Mode=Secondary1 ∧ Battery1.Double ∧ Battery2.Offline
- toS1: Mode=Secondary2 ∧ Battery1.Offline ∧ Battery1.Double

thus they drive the change in operational mode of the batteries.

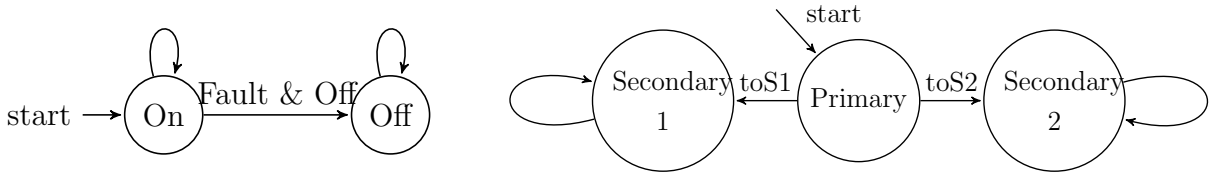

Figure 4.7: Generator (Left) and Switch (Right) LTS

Figure 4.8 shows two slightly more complex components: the sensor and the device. The sensor periodically outputs a good or a bad reading depending on the state it is in. Notice that the transition from a good to a bad state can occur either because of a fault (Wrong Output in Figure 4.6) or because the battery connected to the sensor has no charge ($Batt.c = 0$), notice, in particular, that both events are not observable. The device instead has two main transitions. The *stay* is defined as $S1.Value = S2.Value \wedge Delta = Zero$, while *degrade* represents a discrepancy in the reading from the sensor that will eventually lead to the device stopping: $(S1.Value \neq S2.Value) \wedge Delta =$*Non-Zero*. The values of the sensors are not observable, but their difference is observable via the *Delta* variable. Intuitively, the device has an intermediate state that works as a buffer, before reaching the final *Off* state.

The most complex component, the battery, is presented in Figure 4.9. Vertical transitions indicate a change in operational mode of the battery.

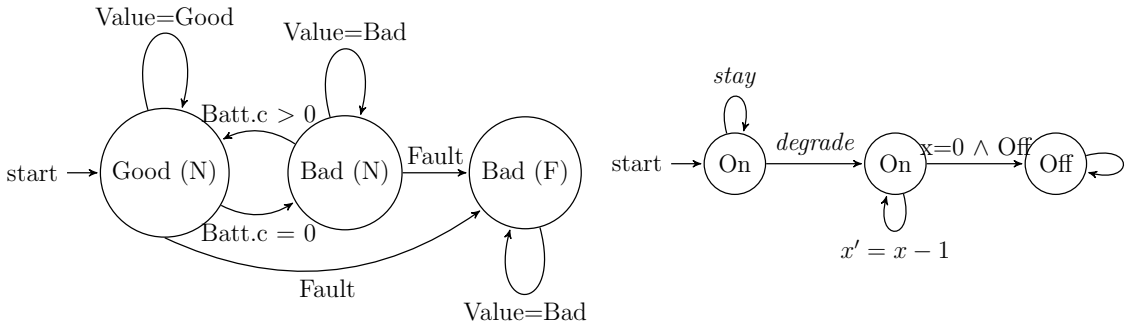

Figure 4.8: Sensor (Left) and Device (Right) LTS

The left half of the LTS indicates that the generator is working and feeding the battery (thus charging it) while the right half shows that the battery is not charging. Additionally, the two central columns describe the faulty behavior of the battery. This information is represented also in each state. Each state has an additional self-loop (not in the picture) denoting the update of the charge of the battery, following the update rule:

$$charge' = (charge + recharge - (load + leak)) \bmod C$$

where C is the capacity, and the other variables depend on the state:

1. Charging: $recharge = 1$, Not Charging: $recharge = 0$
2. Primary: $load = 1$, Offline: $load = 0$, Double: $load = 2$
3. Nominal: $leak = 0$, Faulty: $leak = 2$

Thus the charge of the battery can change from $+1$ (Nominal, Offline, Charging) to -4 (Faulty, Double, Not Charging), while staying within the bound $[0, Capacity]$.

Every time the update of the charge causes the charge to pass a threshold, the transition raises the observable event: Low, Mid, $High$. These events indicate when the charge of the battery is above 20%, 50% and 80%. All other transitions are not observable. These transitions have been omitted from the figure to make it more readable.

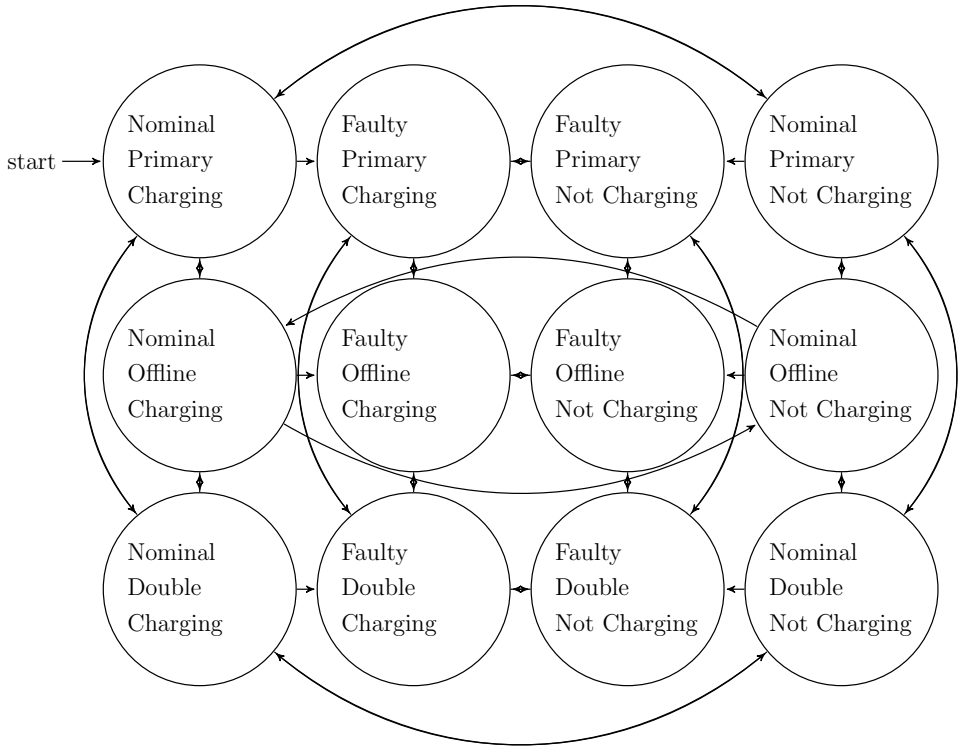

Figure 4.9: Battery LTS

4.6 Chapter Summary

In this chapter, we discussed the general setting of FDIR design in which this thesis is set. In particular, we described the process for modeling the plant, the FDI, FR and FDIRConf. Key concepts in the interaction between the FDIR and the system were introduced: observability requirements, synchronicity and recall. We provided formal definitions for all those concepts, and presented the Battery Sensor System running example.

Chapter 5

Formal Specification of FDI

To be able to build, verify and certify an FDI, we first need to provide a clear definition of what we want the FDI to do. As we discussed in the previous Chapter, the FDI is a function mapping *observations* to *alarms*. In this Chapter, we provide a formal definition of alarm, and show how to specify what we are interested in monitoring (diagnosis condition – Section 5.1), delays in the raising of alarms (alarm condition – Section 5.2), and how many observations the diagnoser can use (recall – Section 5.3). This will provide us with a clear characterization of the FDI, and open the problem of whether an FDI that satisfies the given specification even exists (diagnosability – Section 5.4). Moreover, we will show that when considering delays, we might have multiple behaviors of the FDI that satisfy the same specification, and how to limit this (maximality – Section 5.6). Finally, we discuss under which operational constraints the FDI is supposed to work (context – Section 5.7).

The contribution of this chapter is to formalize the key concepts needed for the specification of alarms. This extends [34] by showing how to deal with both bounded and perfect recall in an unified setting, and by formalizing the concept of Context.

5.1 Diagnosis Conditions

The first element for the specification of the FDI requirements is given by the conditions that must be monitored. The literature usually focuses on detection and identification, which are the two extreme cases of the diagnosis problem. The *detection* task is the problem of understanding when (at least) one fault has occurred. The *identification* task tries to understand exactly which fault has occurred. In the Battery Sensor System (BSS) every component can fail. Therefore the detection problem boils down to knowing that at least one of the generators, batteries or sensors is experiencing a fault. For identification, instead, we are interested in knowing whether a specific fault, (e.g., $G1_{Off}$) occurred.

Between these two cases there can be intermediate ones: we might want to restrict the detection to a particular sub-system, or identification among two similar faults might not be of interest. For example, we might not be interested in distinguishing whether $G1_{Off}$ or $B1_{Leak}$ occurred, as long as we know that there is a problem in the power-supply chain.

FDI components are generally used to recognize faults. However, there is no reason to restrict our interest to faults. Recovery procedures can differ depending on other non-observable conditions of the plant. For example, we might want to estimate the charge level of a battery, or its discharge rate.

We call the condition of the plant to be monitored *diagnosis condition*, denoted by β. We assume that for any point in time along a trace execution of the plant (and therefore also of the system), β is either true or false based on what happened before that time point. Therefore, β can be an atomic condition (including faults), a sequence of atomic conditions, or Boolean combination thereof. If β is a fault, the fault must be identified; if β is a disjunction of faults, it suffices to perform the detection, without

identifying the exact fault.

Diagnosis Condition	**Definition**
$\beta_{Generator1}$	$G1_{Off}$
$\beta_{Battery1}$	$B1_{Leak}$
β_{PSU1}	$G1_{Off} \vee B1_{Leak}$
$\beta_{Batteries}$	$B1_{Leak} \vee B2_{Leak}$
$\beta_{Sensor1}$	$S1_{WO}$
$\beta_{Sensors}$	$S1_{WO} \vee S2_{WO}$
β_{BS}	$(S1_{WO} \vee S2_{WO}) \vee (B1_{Leak} \wedge B2_{Leak})$
β_{Seq}	$(B1_{Charge} < B2_{Charge}) \wedge O(B1_{Charge} \geq B2_{Charge})$
$\beta_{Charging}$	$Y(B1_{Charge} \leq 0) \wedge (B1_{Charge} > 0)$
$\beta_{Depleted}$	$(B1_{Charge} = 0) \vee (B2_{Charge} = 0)$

Figure 5.1: Diagnosis conditions for the Battery-Sensor System

Figure 5.1 shows several examples of diagnosis conditions for the BSS. Conditions might be complex: e.g. knowing if the Battery-Sensor block is working (β_{BS}) or knowing some information on the evolution of the system (β_{Seq}, $\beta_{Charging}$). In the example, we use LTL operators to define those diagnosis conditions, but in general, we require that a diagnosis condition can be evaluated on a point in a trace by only looking at the trace prefix.

5.2 Delay and Alarm Conditions

The second element of the specification of FDI requirements is the relation between a diagnosis condition and the raising of an alarm. This also leads to the definition of when the FDI is correct and complete with regard to a set of alarms.

An *alarm condition* is composed of two parts: the diagnosis condition and the delay. The delay relates the time between the occurrence of the diagnosis condition and the corresponding alarm. Although it might

be acceptable that the occurrence of a fault can go undetected for a certain amount of time, it is important to specify clearly how long this interval can be. An alarm condition is a property of the system composed by the plant and the diagnoser, since it relates a condition of the plant with an alarm of the diagnoser. Thus, when we say that a diagnoser D of P satisfies an alarm condition, we mean that the traces of the combined system $D \otimes P$ satisfy it. Interaction with industrial experts led us to identify three patterns of *alarm conditions*, which we denote by $\textsc{ExactDel}(A, \beta, d)$, $\textsc{BoundDel}(A, \beta, d)$, and $\textsc{FiniteDel}(A, \beta)$:

1. $\textsc{ExactDel}(A, \beta, d)$ specifies that whenever β is true, A must be triggered exactly d steps later and A can be triggered only if d steps earlier β was true; formally, for any trace σ of the system:

   (*Completeness*) if β is true along σ at the time point i, then $\llcorner A \lrcorner$ is true in $\sigma[i + d]$;

   (*Correctness*) if $\llcorner A \lrcorner$ is true in $\sigma[i]$, then β must be true in $\sigma[i - d]$.

2. $\textsc{BoundDel}(A, \beta, d)$ specifies that whenever β is true, A must be triggered within the next d steps and A can be triggered only if β was true within the previous d steps; formally, for any trace σ of the system:

   (*Completeness*) if β is true along σ at the time point i then $\llcorner A \lrcorner$ is true in $\sigma[j]$, for some $i \leq j \leq i + d$;

   (*Correctness*) if $\llcorner A \lrcorner$ is true in $\sigma[i]$, then β must be true in $\sigma[j']$ for some $i - d \leq j' \leq i$.

3. $\textsc{FiniteDel}(A, \beta)$ specifies that whenever β is true, A must be triggered in a later step and A can be triggered only if β was true in some previous step; formally, for any trace σ of the system:

(*Completeness*) if β is true along σ at the time point i then $\llcorner A \lrcorner$ is true in $\sigma[j]$ for some $j \geq i$;

(*Correctness*) if $\llcorner A \lrcorner$ is true in $\sigma[i]$, then β must be true along σ in some time point between 0 and i.

Figure 5.2 provides an example of admissible responses for the various alarms to the occurrences of the same diagnosis condition β; note how in the case of BoundDel$(A, \beta, 4)$ the alarm can be triggered at any point as long as it is within the next 4 time-steps. Since A is a state variable and the diagnoser changes it only in response to synchronizations with the plant, every rising and falling edge of the alarm in the figure corresponds to an observation point.

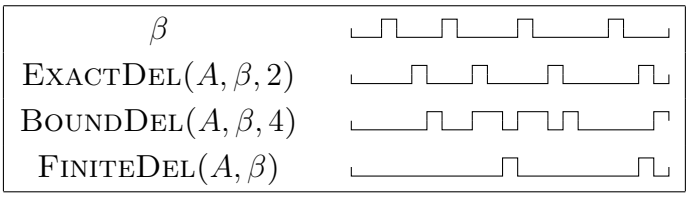

Figure 5.2: Examples of alarm responses to the diagnosis condition β.

Pattern	**Description**
ExactDel$(PSU1_{Exact_i}, \beta_{PSU1}, i)$	Detect if the PSU 1 (Generator 1 + Battery 1) is broken, in order to switch to secondary mode
BoundDel$(PSU1_{Bound}, \beta_{PSU1}, C)$	Detect if the PSU (Generator 1 + Battery 1) was broken within the bound, in order to switch to secondary mode
BoundDel(BS, β_{BS}, DC)	Detect if the whole Battery-Sensor block is working incorrectly, in order to replace it
FiniteDel$(Discharged, \beta_{Depleted})$	Detect if any of the battery was ever completely discharged

Figure 5.3: Example Specification for the Battery-Sensor System

Figure 5.3 contains a simple specification for our running example. There are two types of PSU (Power Supply Unit) alarms (that can be similarly defined for PSU 2). The first one defines multiple alarms, each having a different delay i. Let us assume that each battery has a capacity C of 10, and that this provides us with a delay of at most 10 time-units. We can instantiate 10 alarms one for each $i \in [0, 10]$. Ideally, we want to detect the exact moment in which the PSU stopped working. However, it might not be possible to know the precise moment due to the limited amount of information available (non-diagnosability). Therefore, we define a weaker version of the alarm ($PSU1_{Bound}$), in which we say that within the time-bound provided by the battery capacity (C) we want to know if the PSU stopped working. For most alarms, we specify what recovery can be applied to address the problem. In this way, our process of defining the alarms of interest is driven by the recovery procedures available. If there is no automated recovery for a given situation, time-bounds might not be relevant anymore. Therefore, we use alarms to collect information on the historical state of the system (e.g., *Discharged* alarm); notice, in fact, that FiniteDel alarm have a permanent behavior, i.e., they can never be turned off.

5.3 Diagnoser Recall

A diagnoser might have constraints on the amount of information that it can remember and use for reasoning. We consider two cases: *perfect recall* and *bounded recall*. To capture the concept of recall, and the amount of observations performed by each observer, we use the concept of observable trace of a partially observable transition system as *ground-truth* of what can be observed, and derive all other possible observable traces from it.

We define a family of observation functions that can either be perfect

recall or bounded recall, and are parametric w.r.t. the set of observable events.

Definition 11 (Reduced Observable Trace)**.** *Given an observable trace* $\omega = e_1, \cdots, e_k$ *computed over the set of events* E_O*, a recall* $\mathcal{R} \in [0..\infty]$*, and the assumption that* $\infty - 1 = \infty$*, we have that* $obs^{\mathcal{R}}_{E_D}$ *is defined as:*

- $\mathcal{R} > 0$*:*
  - $e_k \in E_D$*:* $obs^{\mathcal{R}}_{E_D}(e_1, \cdots, e_k) = obs^{\mathcal{R}-1}_{E_D}(e_1, ..., e_{k-1})e_k$
  - $e_k \notin E_D$*:* $obs^{\mathcal{R}}_{E_D}(e_1, \cdots, e_k) = obs^{\mathcal{R}}_{E_D}(e_1, ..., e_{k-1})$
- $\mathcal{R} = 0$*:*
  - $e_k \in E_D$*:* $obs^{\mathcal{R}}_{E_D}(e_1, \cdots, e_k) = e_k$
  - $e_k \notin E_D$*:* $obs^{\mathcal{R}}_{E_D}(e_1, \cdots, e_k) = \epsilon$

with the base case $obs^{\mathcal{R}}_{E_D}(\epsilon) = \epsilon$*.*

For perfect recall ($\mathcal{R} = \infty$), we lose information only if we reduce the set of observable events. For bounded recall, instead, we reduce the trace to the latest $\mathcal{R}$ observations. It might happen that there are not enough observations, thus we obtain an observable trace that is less than $\mathcal{R}$ events long. We write $obs^{\infty}_{E_D}$ to indicate a *perfect recall* observation function w.r.t. the events set E_D; we write $obs^{n}_{E_D}$ to indicate the *bounded recall* observation function w.r.t. the events set E_D and with n recall steps.

If we do not specify the recall, we intend perfect, and if we do not specify the set of events, we consider all the observable events of the plant. This also matches the fact that the best possible diagnoser is the one with access to the most information and perfect recall.

For every trace σ of a plant, we can compute the observable trace $\omega = obs(\sigma)$. Therefore, for simplicity, we extend the definition of $obs^{\mathcal{R}}_{E_D}$ in order to be applicable on traces, and write $obs^{\mathcal{R}}_{E_D}(\sigma)$ instead of $obs^{\mathcal{R}}_{E_D}(obs(\sigma))$.

The recall information is central to the definition of *observationally equivalent* traces.

Definition 12 (Observational Equivalence)**.** *Given two traces and points* (σ_1, i) *and* (σ_2, j)*, a set of observable events* E_O *and a recall* $\mathcal{R}$*, we say that* $ObsEq^{\mathcal{R}}_{E_O}((\sigma_1, i), (\sigma_2, j))$ *iff:*

- $ObsPoint(\sigma_1, i)$ *iff* $ObsPoint(\sigma_2, j)$*, and*

- $obs^{\mathcal{R}}_{E_O}(\sigma_1^i) = obs^{\mathcal{R}}_{E_O}(\sigma_2^j)$.

We write *ObsEq* if the set of events and the type of recall are clear from the context. This abstract definition of observational equivalence allows us to consider multiple degrees of observability and recall within the same framework.

5.4 System Diagnosability

Given an alarm condition, we need to know whether it is possible to build a diagnoser for it. In fact, there is no reason in having a specification that cannot be realized. This property is called *diagnosability* and was introduced in [137]. We adapt the concept of diagnosability for the different types of alarm conditions.

Definition 13. *Given a plant* P*, a diagnosis condition* β*, a recall* $\mathcal{R}$ *and a set of observables* E_O*, we say that* $\textsc{ExactDel}(A, \beta, d)$ *is* system diagnosable *in* P *iff for all* (σ_1, i) *s.t.* $\sigma_1, i \models \beta$ *then* $ObsPoint(\sigma_1, i + d)$ *and for all* (σ_2, j)*, if* $ObsEq^{\mathcal{R}}_{E_O}((\sigma_1, i + d), (\sigma_2, j + d))$*, then* $\sigma_2, j \models \beta$.

Therefore, an exact-delay alarm condition is not diagnosable in P iff either there is no synchronization after d steps (note that this is not possible in the synchronous case) or there exists a pair of traces σ_1 and σ_2 such that

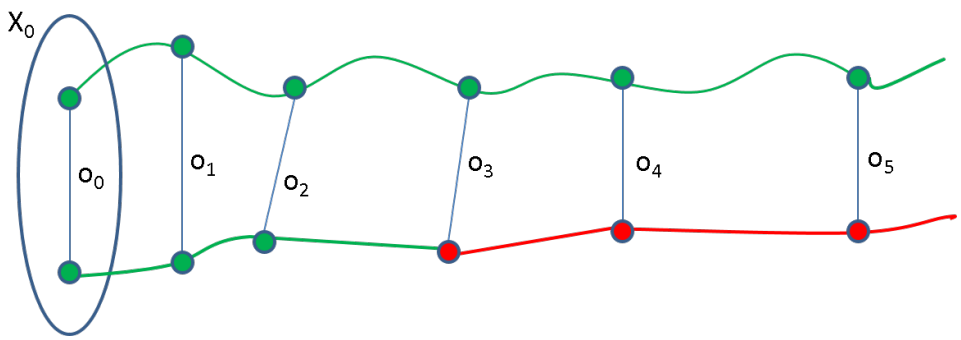

Figure 5.4: Critical Pair: A fault occurs but observations match a nominal execution.

for some $i, j \geq 0$, $\sigma_1, i \models \beta$, $ObsEq((\sigma_1, i+d), (\sigma_2, j+d))$, and $\sigma_2, j \not\models \beta$. We call such a pair a *critical pair*.

The idea of a critical pair is that a diagnoser that can only see the observable part of the trace, cannot know whether the system is executing the trace σ_1 or σ_2, as shown in Figure 5.4 (for a given recall and observables set). Notice that this uncertainty is only in the diagnoser's mind, and the actual plant must be in either one or the other. Extending the observables set or increasing the recall might disambiguate the two traces.

Definition 14. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, and a set of observables E_O, we say that* BoundDel(A, β, d) *is* system diagnosable *in P iff forall (σ_1, i) s.t. $(\sigma_1, i) \models \beta$ there exists k s.t. $i \leq k \leq i + d$, $ObsPoint(\sigma_1, k)$ and for all (σ_2, l), if $ObsEq^{\mathcal{R}}_{E_O}((\sigma_1, k), (\sigma_2, l))$, then there exists j s.t. $l - d \leq j \leq l$ and $(\sigma_2, j) \models \beta$.*

Intuitively, k and l denote points that are observationally equivalent, i and j denote the states where the condition occurred, and their relation is such that i and j do not occur more than d steps away from each other.

This definition takes into account occurrences of β that happened before i. Indeed, we need to check occurrences up to d states before and after i. Consider the two traces $\sigma_1 = apbqc$ and $\sigma_2 = aqbpc$, where a, b, c are observable events, and $\beta = p$. We can see that we can justify p in σ_1 by

looking at the occurrence of p in σ_2 that is in the future. However, we cannot justify the p in σ_2 by just looking in the future, but we need to look in the past.

Differently from the classical definition of [137], we use the delay to limit the scope of comparison of the traces of the critical pair. This means that we do not require the diagnosis condition to be permanent but allow transient conditions (e.g., transient faults). Moreover, we can deal with multiple occurrences of the same condition transparently.

Sampath's definition of diagnosability can be obtained as a special case of Definition 14:

Definition 15. *(Sampath's Diagnosability [137]) Given a plant P and a diagnosis condition β, we say that β is diagnosable in P iff there exists d s.t. for all (σ, i), (σ_2, l), $k \geq i + d$ if $\sigma_1, i \models \beta$ and $obs(\sigma_2^l) = obs(\sigma_1^k)$ then there exists $j \leq l$ s.t. $(\sigma_2, j) \models \beta$.*

In [137] (specifically in Section II.A), Sampath et al. also assume that there are no cycles of unobservable events. This means that there is a d_u s.t. for all σ, i s.t. $(\sigma, i) \models \beta$ then there exists k s.t. $0 \leq k \leq d_u$ and $ObsPoint(\sigma, i + k)$. Moreover, the definition of *obs* used there requires perfect recall and observability over all observable events.

Theorem 1. *Let P 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 15) in P iff there exists d such that* BoundDel(A, Op, d) *is system diagnosable in P.*

Proof. $\Rightarrow$) Assume that p is diagnosable in P. Consider a trace σ_1 such that for some $i \geq 0$, $(\sigma_1, i) \models Op$. Then, for some $0 \leq i' \leq i$, $(\sigma_1, i') \models p$. By assumption, we know that there is a d s.t. for all $k \geq i' + d$ and any trace σ_2 and point l such that $obs(\sigma_2^l) = obs(\sigma_1^k)$ then $(\sigma_2, j') \models p$ for some j', $j' \leq l$. Then $(\sigma_2, j) \models Op$ for all $j \geq j'$.

Since this holds for any k and l, it holds also for the k and l that are observation points for σ_1 and σ_2. Let $d' = d + n_u$. Then there exists $k' < d'$ such that $ObsPoint(\sigma_1, i + k')$ and for all (σ_2, l) such that $ObsEq^{\infty}_{E_O}((\sigma_1, k'), (\sigma_2, l))$ then $(\sigma_2, j') \models p$ for some j', $j' \leq l$. We can conclude that $\textsc{BoundDel}(A, Op, d')$ is system diagnosable in P.

$\Leftarrow$) Assume that $\textsc{BoundDel}(A, Op, d)$ is system diagnosable in P. Consider a trace σ_1 such that for some $i \geq 0$ $(\sigma_1, i) \models p$. Then $(\sigma_1, i) \models Op$. By assumption, there exists k, $i \leq k \leq i + d$ such that $ObsPoint(\sigma_1, k)$ and, for any trace σ_2 and point l such that $ObsEq^{\infty}_{E_O}((\sigma_1, k), (\sigma_2, l))$ then $\sigma_2, j \models Op$ for some $l - d \leq j \leq l$. Let us consider σ_2 and l such that $obs(\sigma_2^l) = obs(\sigma_1^k)$. Then for some $l' \leq l$ we have that $ObsPoint(\sigma_2, l')$ and therefore $ObsEq^{\infty}_{E_O}((\sigma_1, k), (\sigma_2, l'))$. Then $(\sigma_2, j) \models Op$ for some $l - d \leq j \leq l$. Thus $(\sigma_2, j') \models p$ for some $j' \leq j$ and P is diagnosable.

□

It is important to avoid confusing the delay of the alarm with the amount of recall available to the diagnoser. It is possible for a diagnoser to raise an alarm with a delay that is longer than its recall. Let us say that a process in the plant requires exactly 100 time-steps to complete. Upon seeing the event signaling the completion of the process, the diagnoser knows (even with bounded recall 0) that the process started 100 time-steps before. Similarly, having a long (or perfect) recall does not guarantee that the diagnoser will be able to diagnose all alarms that have a short delay.

If we do not want to include any requirement on delay between the diagnosis condition occurrence and the raising of the alarm, we use finite delay alarm conditions.

Definition 16. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, and a set of observables E_O, we say that* FiniteDel(A, β) *is* system diagnosable *in P iff for all (σ_1, i) s.t. $(\sigma_1, i) \models \beta$ then there exist $k \geq i$ s.t. $ObsPoint(\sigma_1, k)$ and for all (σ_2, l) if $ObsEq^{\mathcal{R}}_{E_O}((\sigma_1, k), (\sigma_2, l))$ then there exists $j \leq l$ $(\sigma_2, j) \models \beta$.*

In bounded delay (and in Sampath) we consider a delay d that works for any trace. In finite delay, instead, we swap the quantifiers and *for each* trace we pick a (potentially different) delay d that is enough to exclude all critical pairs. This is a weaker form of diagnosability, and therefore there are alarm conditions that are not bounded delay system diagnosable but that are finite delay system diagnosable. The intuition is that in some cases we can extend a critical pair at will. For example, imagine a system that at each transition reduces the value of a variable x (i.e., $x' = x - 1$). Let us now assume that we can only observe $x = 0$. The diagnosis condition is $\beta := (x = (init_x/2))$, i.e. x has half its initial value. We can see that whenever we reach $x = 0$ we know that previously β was met. However, how many steps ago this happened depends on the initial value of x (that is not observable!). For an infinite state system we can always pick a trace with an initial value of x that is slightly bigger.

The following theorem shows that if a component satisfies the diagnoser specification then the monitored plant must be diagnosable for that specification. In Chapter 8 (Synthesis) we will show also the converse, i.e., if the specification is diagnosable then a diagnoser exists.

Theorem 2. *Let D be a diagnoser for P. If D satisfies an alarm condition then the alarm condition is system diagnosable in P.*

Proof. By contradiction, suppose ExactDel(A, β, d) is not system diagnosable in P. Then either there exists a trace σ_1 with $\sigma_1, i \models \beta$ for some i such that $ObsPoint(\sigma_1, j)$ is false for all $j \geq i$ or there exists a critical

pair. In the first case, A is not triggered and the diagnoser is not complete. Suppose there exists a critical pair of traces σ_1 and σ_2, i.e., for some $i, j \geq 0$ $(\sigma_1, i) \models \beta$, $ObsPoint(\sigma_1, i+d)$, $ObsEq((\sigma_1, i+d), (\sigma_2, j+d))$, and $\sigma_2, j \not\models \beta$. Since D is deterministic, $D(\sigma_1)$ and $D(\sigma_2)$ have a common prefix compatible with $obs(\sigma_1^{i+d}) = obs(\sigma_2^{j+d})$. If the diagnoser is complete then A is triggered in $D(\sigma_1) \otimes \sigma_1$ at position $i+d$, and so also in $D(\sigma_2) \otimes \sigma_2$ at position $j+d$, but in this way the diagnoser is not correct, which is a contradiction. If the diagnoser is correct, then A is not triggered in $D(\sigma_2) \otimes \sigma_2$ at position $j+d$, but so neither in $D(\sigma_1) \otimes \sigma_1$ at position $i+d$, but in this way the diagnoser is not complete, which is a contradiction.

Similarly, for $\textsc{FiniteDel}(A, \beta)$ and $\textsc{BoundDel}(A, \beta, d)$. □

5.5 Trace Diagnosability

System diagnosability is defined as a global property of the plant. This definition of diagnosability might be stronger than necessary. Imagine the situation in which there is a critical pair and after removing this critical pair from the possible executions of the system, our system becomes diagnosable. This suggests that the system was *almost* diagnosable, and an ideal diagnoser would be able to perform a correct diagnosis in all the cases except one: the one represented by the critical pair. We formalize this idea by redefining the problem of diagnosability from a global property expressed on the plant, to a local property expressed on points of single traces.

Definition 17. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, a set of observables E_O, and a trace σ_1 such that for some $i \geq 0$ $(\sigma_1, i) \models \beta$, we say that* $\textsc{ExactDel}(A, \beta, d)$ *is* trace diagnosable in (σ_1, i) *iff $ObsPoint(\sigma_1, i+d)$ and for any trace σ_2, for all $j \geq 0$ such that $ObsEq^{\mathcal{R}}_{E_O}((\sigma_1, i+d), (\sigma_2, j+d))$, $(\sigma_2, j) \models \beta$.*

Definition 18. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, a set of observables E_O, and a trace σ_1 such that for some $i \geq 0$ $(\sigma_1, i) \models \beta$, we say that* BoundDel$(A, \beta, d)$ *is* trace diagnosable in (σ_1, i) *iff there exists k s.t. $i \leq k \leq i + d$, $ObsPoint(\sigma_1, k)$, and for any (σ_2, l) if $ObsEq^{\mathcal{R}}_{E_O}((\sigma_1, k), (\sigma_2, l))$, then there exists j s.t. $l - d \leq k \leq l$ and $(\sigma_2, j) \models \beta$.*

Definition 19. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, a set of observables E_O, and a trace σ_1 such that for some $i \geq 0$, $(\sigma_1, i) \models \beta$, we say that* FiniteDel$(A, \beta)$ *is* trace diagnosable in (σ_1, i) *iff there exists $k \geq i$ s.t. $ObsPoint(\sigma_1, k)$ and for all (σ_2, l) if $ObsEq^{\mathcal{R}}_{E_O}((\sigma_1, k), (\sigma_2, l))$, then there exists $j \leq l$ and $(\sigma_2, j) \models \beta$.*

A specification that is trace diagnosable in a plant along all points of all traces is diagnosable in the classical sense, and we say it is *system* diagnosable. The concept of trace diagnosability does not impose any specific behavior to the diagnoser. However, it is an important concept that allows us to better characterize and understand the specification and the system. While this is a weaker definition than system diagnosability, it might be the case that it is not satisfied by the plant. Therefore, we have 3 degrees of diagnosability that can be defined in terms of trace diagnosability:

1. System Diagnosable: $\forall(\sigma, i)$. trace diagnosable in (σ, i);
2. Trace Diagnosable: $\exists(\sigma, i)$. trace diagnosable in (σ, i);
3. Non-Diagnosable: $\forall(\sigma, i)$. not trace diagnosable in (σ, i).

5.6 Maximality

As shown in Figure 5.2, bounded- and finite-delay alarms are correct if they are raised within the time-bound. However, there are several possible variations of the same alarm in which the alarm is active in different

instants or for different periods. We address this problem by introducing the concept of *maximality*. Intuitively, a maximal diagnoser is required to raise the alarms as soon as possible and as long as possible (without violating the correctness condition). Figure 5.5 shows the occurrence of the diagnosis condition, and both a maximal and non maximal alarm, for a bounded delay specification. Notice that the non maximal alarm raises and lowers the alarm earlier than the maximal one. This behavior is still correct, but introduces non-determinism in the behavior of the diagnoser.

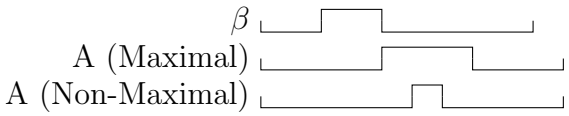

Figure 5.5: Maximal and Non-Maximal traces for Bounded Delay Alarm

Definition 20 (Maximality)**.** *D is a maximal diagnoser for an alarm condition with alarm A in P iff for every trace σ_P of P, $D(\sigma_P)$ contains the maximum number of observable points i such that $\big(D(\sigma_P), i\big) \models A$; that is, if $\big(D(\sigma_P), i\big) \not\models A$, then there does not exist another correct diagnoser D' of P such that $\big(D'(\sigma_P), i\big) \models A$.*

By forcing a diagnoser to be maximal, we precisely characterize when a diagnoser should raise the alarm. This makes it possible to compare two different implementations of the same diagnoser: since the diagnoser is a deterministic machine, we can check for equivalence of the input/output behavior. This is particularly useful if we are interested in comparing manually designed against automatically synthesized diagnosers, or if we are using some code optimization/minimization techniques.

Theorem 3. *Let D_1 and D_2 have the same recall, access to the same set of observables, and be correct and maximal w.r.t. the alarm condition A in*

P. For each observable trace σ_P the alarm sequences of the two matching traces coincide, i.e., $\sigma_{D_1}|_A = \sigma_{D_2}|_A$

Proof. Let us assume that there is a state at time i in σ_{D_1} s.t. the two diagnosers behave differently, i.e., $A_1 \neq A_2$, where $A_1 = \sigma_{D_1}|_A[i]$ and $A_2 = \sigma_{D_2}|_A[i]$. Let us assume that A_1 is true while A_2 is false (the other case can be shown by symmetry). Since D_1 is correct (by assumption), it means that the diagnosis condition occurred and also D_2 needs to raise the alarm. Since D_2 is maximal, we know that if it did not raise the alarm either the condition is not diagnosable or D_2 is incorrect. D_2 is correct by assumption, and the condition is diagnosable, because it is diagnosable for D_1 that has the same observables and recall as D_2. □

5.7 Context

The use of Sampath's definition of diagnosability (and similarly of system diagnosability in this work), has lead to the problem that most systems are not (system) diagnosable. This is usually the result of some uncontrollable situation that is needed in order to disambiguate a critical pair. A typical example is the case of a broken light-bulb connected to a light switch. Unless we try to turn the light on, we are unable to understand whether it is broken. If the action of turning the light on is free to never happen, we have a non (system) diagnosable situation. In [137] the concept of *indicator events* is used to deal with this problem. Indicator events are observable events that are required to occur after the fault, to limit the scope of the diagnosability analysis (called I-diagnosability in [137]). In our example, the indicator event would be the toggling of the light-switch. The definition of the indicator events is quite restrictive, since it requires the indicator events to be known up-front, and associated with the faults. Our definition of trace diagnosability solves this problem, since the designer does not need

to specify which are the indicators events. If the action of switching the light on is observable, the system is trace diagnosable. In particular, the traces in which we switch on the light are diagnosable.

However, if the action of switching the light is not observable, the system is non-diagnosable (neither system or trace). Clearly, if we add the assumption that every 10 seconds somebody will turn on the light, then we end up with a system diagnosable situation.

It is quite common to describe the plant with all its possible behaviors, although the operational context in which the plant will work might restrict those behaviors. Two common examples are *qualitative considerations* and *contract based design*.

In the design of safety critical systems, we use components and architectures that have a very low failure rate. Therefore, the chances of several components failing at the same time becomes extremely unlikely. In this sense, we can use these *qualitative considerations* in order to constraint the execution of our system, and exclude traces in which, for example, three or more faults occur. By doing this we are removing traces from the system and potentially improving its diagnosability.

Another example of operational context is given by *contract based design* [57]. In this design methodology, each component is equipped with some assumptions it expects from (and needs to guarantee to) the components to which it is connected. Contract based design tries to encourage components re-use. Therefore, we want to model the component in a general way, and then check whether the behavior of the other components can restrict the operational context. The example of the light being switched on every 10 seconds can be seen as a contract based design example, in which the environment of the component guarantees that the switching signal will be issued every 10 seconds.

The *operational context* (context) of the plant P is a subset of the traces

of the plant: $\mathcal{C} \subseteq \Sigma(P)$. This is equivalent to defining a modified plant P' that has the given language. However, from an operational point of view, it is usually simpler to keep the same model and add external constraints. Given a context $\mathcal{C}$, the concepts of diagnosability, correctness, completeness, and maximality can be refined w.r.t. $\mathcal{C}$, by simply considering only traces in $\mathcal{C}$ instead of traces in $\Sigma(P)$.

5.8 Chapter Summary

In this chapter, we discussed how to formally specify alarms for the FDI. We defined the concepts of diagnosis condition, delay, and alarm condition. In order to capture the reasoning capabilities of the diagnoser, we formally characterize the relation between recall and observability. In turn, this required us to introduce the concept of diagnosability, in order to understand when an alarm specification can be satisfied. We discussed the classical definition of system diagnosability and, to overcome its limitations, we introduced the concept of trace diagnosability. In order to guarantee a deterministic input/output behavior of the diagnoser, we introduce the concept of maximality. Finally, we discuss how to impose additional operational constraints on the FDI by using a context.

Chapter 6

ASL_K

In this chapter, we present the Alarm Specification Language with Epistemic operators (ASL_K). The goal of ASL_K is to allow designers to formally specify the alarms conditions covering all relevant aspects such as completeness, correctness, delays, maximality, and context. The semantics of each alarm specification can then be encoded into temporal epistemic logic. This allows us to apply automated reasoning techniques on the specification, and verify whether a model satisfies a given specification (model-checking).

We start by describing how to capture diagnosis and alarm conditions using LTL with past operators (Section 6.1). Afterwards, we remark that trace diagnosability and maximality cannot be captured by using a formalization based on LTL. To capture these two concepts, we rely on the temporal epistemic logic KL_1 (Section 6.2). The intuition is that this logic enables us to reason on set of observationally equivalent traces instead that on single traces (like in LTL). We show how this logic can be used to specify diagnosability, define requirements for non-diagnosable cases and express the concept of maximality. All these information are combined within the specification (Section 6.3). The availability of a logical grounding makes it possible to apply automated reasoning to perform validation

and verification, as shown in Section 6.4. In Section 6.5, we provide examples of ASL$_K$ specifications.

The contributions of this chapter are ASL$_K$, its mapping on top of temporal epistemic logic, and its application on several examples. This extends the results presented in [34] by considering the problem of recall. Extended examples from both [34] and [32] are also presented.

6.1 Diagnosis and Alarm Conditions as LTL

Table 6.1: Alarm conditions as LTL (ASL): *Correctness* and *Completeness*

Alarm Condition	**Correctness**	**Completeness**
ExactDel(A, β, d)	$G(\llcorner A \lrcorner \rightarrow Y^d \beta)$	$G(\beta \rightarrow X^d \llcorner A \lrcorner)$
BoundDel(A, β, d)	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta)$	$G(\beta \rightarrow F^{\leq d} \llcorner A \lrcorner)$
FiniteDel(A, β)	$G(\llcorner A \lrcorner \rightarrow O\beta)$	$G(\beta \rightarrow F \llcorner A \lrcorner)$

Let $\mathcal{P}$ be a set of propositions representing either faults, events or elementary conditions for the diagnosis. Let $p \in \mathcal{P}$, then the set $\mathcal{D}_\mathcal{P}$ of *diagnosis conditions* over $\mathcal{P}$ is any formula β built with the following rule:

$$\beta ::= p \mid \beta \wedge \beta \mid \neg\beta \mid \beta S \beta \mid O\beta \mid Y\beta$$

In practice, we are interested in safety properties, (i.e., in prefix-closed properties [8]) therefore the definition of β could be slightly extended to include other LTL operators, or use a different formalism for safety properties. We focus on past LTL for clarity, simplicity, and availability of tools.

Table 6.1 gives the LTL characterization of the *Alarm Specification Language* (ASL). The name of each alarm condition (ExactDel, BoundDel, and FiniteDel) is associated with the LTL encoding of the concepts of correctness and completeness:

Correctness the first conjunct, says that whenever the diagnoser raises an alarm, the fault must have occurred;

Completeness the second conjunct, encodes that whenever the fault occurs, the alarm will be raised.

In the following, for simplicity, we abuse notation and indicate with φ both the alarm condition and the associated LTL; for an alarm condition φ, we denote by A_φ the associated alarm variable A, and with $\tau(\varphi)$ the following formulas, that characterize the *temporal* behavior of the delay:

- $\tau(\varphi) = Y^d\beta$ for $\varphi = \textsc{ExactDel}(A, \beta, d)$;

- $\tau(\varphi) = O^{\leq d}\beta$ for $\varphi = \textsc{BoundDel}(A, \beta, d)$;

- $\tau(\varphi) = O\beta$ for $\varphi = \textsc{FiniteDel}(A, \beta)$.

when clear from the context, we use A and τ instead of A_φ and $\tau(\varphi)$.

6.2 Diagnosability and Maximality as KL_1

To know if the diagnoser is complete and correct, we only need to look at the current trace. Therefore, LTL is sufficient to capture those concepts. However, the concepts of diagnosability and maximality require us to consider sets of observationally equivalent traces. For this reason, LTL is not sufficient anymore, and we need a more expressive logic. In particular, we decided to use the temporal epistemic logic KL_1 (Section 2.4).

The semantics of K_i is given in terms of the accessibility relation $\sim_i$ for agent i. Given a recall $\mathcal{R}$ and a set of observable E_o for the diagnoser D_i, the accessibility relation $\sim_i$ coincides with the observational equivalence relation $ObsEq^{\mathcal{R}}_{E_O}$ (i.e., $\sim_i \equiv ObsEq^{\mathcal{R}}_{E_O}$ – Definition 12).

Table 6.2: *Diagnosability*

Alarm Condition	**Diagnosability**
ExactDel(A, β, d)	$G(\beta \rightarrow X^d \llcorner KY^d \beta \lrcorner)$
BoundDel(A, β, d)	$G(\beta \rightarrow F^{\leq d} \llcorner KO^{\leq d} \beta \lrcorner)$
FiniteDel(A, β)	$G(\beta \rightarrow F \llcorner KO \beta \lrcorner)$

Multiple diagnosers $(D_i, \cdots, D_j)$ can co-exist with access to different observables and recall. This allows us to abstract the specification language from the details of observability and recall of the diagnoser, and experiment with different types of diagnosers while keeping the specification unchanged. For example, consider a perfect recall diagnoser D_1 and a bounded recall 5 diagnoser D_2. We can write properties that relate their knowledge: $G(K_1\varphi \rightarrow K_2\varphi)$. In the rest of the chapter we simply write K when the details of the diagnoser are not relevant.

6.2.1 Diagnosability as KL_1

The definition of completeness for ASL holds only if the plant is system diagnosable. In order to weaken this condition, we need to show how to encode diagnosability as a KL_1 property.

This encoding is independent of the recall and observability of the diagnoser, since these information are embedded in the semantics of the modal operator K. Moreover, this provides us with a way of performing both system and trace diagnosability tests.

Table 6.2 shows the encoding of diagnosability as a temporal epistemic property. Intuitively, we say that every time that the diagnosis condition occurs, the diagnoser knows that it has occurred after a certain delay (that depends on the alarm condition type). Due to the asynchronous nature of our setting, we require the knowledge operator to hold during an observation point. In order to test for system diagnosability, we will check

whether the formula holds for all traces of the system, e.g.:

$$P \models G(\beta \rightarrow F_{\llcorner} KO\beta_{\lrcorner})$$

For example, the diagnosability test for $\textsc{ExactDel}(A, \beta, d)$ says that it is always the case that whenever β occurs, exactly d steps afterwards, the diagnoser *knows* β occurred d steps earlier. 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 d steps) in which β did not occur; but this is in contradiction with the definition of diagnosability (Definition 13).

Table 6.3: Non-diagnosability.

Alarm Condition	**Non-Diagnosability**
$\textsc{ExactDel}(A, \beta, d)$	$G(\beta \rightarrow X^d \neg_{\llcorner} KY^d \beta_{\lrcorner})$
$\textsc{BoundDel}(A, \beta, d)$	$G(\beta \rightarrow G^{\leq d} \neg_{\llcorner} KO^{\leq d} \beta_{\lrcorner})$
$\textsc{FiniteDel}(A, \beta)$	$G(\beta \rightarrow G \neg_{\llcorner} KO\beta_{\lrcorner})$

If we obtain a counter-example to the system diagnosability, we are interested in understanding whether the plant is trace diagnosable. This means finding some traces and points that are diagnosable. To do so we try to prove that every trace and point is *not* trace diagnosable by negating the right hand side of the system diagnosability test (Table 6.3):

$$P \models G(\beta \rightarrow G^{\leq d} \neg_{\llcorner} KO^{\leq d} \beta_{\lrcorner})$$

If the plant satisfies the property, it means that the alarm condition is non-diagnosable in every trace and point. A counter-example to the property is a witness of a trace and point that are trace diagnosable.

Finally, note that the formulation of diagnosability is the same for both system diagnosability and trace diagnosability. If we are interested in system diagnosability, we verify that all traces and points satisfy the condition. Whereas, for trace diagnosability, we just check whether a trace in a point satisfies the property.

Table 6.4: Trace Completeness.

Alarm Condition	**Trace Completeness**
$\textsc{ExactDel}(A, \beta, d)$	$G(\llcorner(\beta \wedge X^d \llcorner K Y^d \beta\lrcorner) \rightarrow X^d \llcorner A\lrcorner)$
$\textsc{BoundDel}(A, \beta, d)$	$G(\llcorner(\beta \wedge F^{\leq d} \llcorner K O^{\leq d} \beta\lrcorner) \rightarrow F^{\leq d} \llcorner A\lrcorner)$
$\textsc{FiniteDel}(A, \beta)$	$G(\llcorner(\beta \wedge F \llcorner K O \beta\lrcorner) \rightarrow F \llcorner A\lrcorner)$

Trace Completeness If the alarm condition for the plant is not system diagnosable, we cannot show that the diagnoser is complete using the LTL specification from Table 6.1, since any critical pair would be a counter-example to completeness. Therefore, we propose to use the definition of diagnosability to restrict the completeness on diagnosable traces. Since we are using LTL, we can imagine that we are considering one trace at the time. If we are in a point of a trace that is trace diagnosable, we require the diagnoser to be complete. A counter-example to this property, when checked on the plant and diagnoser, is a trace that is diagnosable but for which the diagnoser did not raise the alarm:

$$D \otimes P \models G((\beta \rightarrow F^{\leq d} \llcorner K O^{\leq d} \beta\lrcorner) \rightarrow (\beta \rightarrow F^{\leq d} \llcorner A\lrcorner))$$

In Table 6.4 we show the formulation of trace completeness, in which we combined completeness and diagnosability. Notice that we simplify the expressions by rewriting the two implications into a conjunction according to the rule: $(A \rightarrow B) \rightarrow (A \rightarrow C) \Rightarrow (A \wedge B) \rightarrow C$.

6.2.2 Maximality as $\boldsymbol{KL}_1$

A maximal diagnoser will raise the alarm as soon as it is possible to know the diagnosis condition, and the alarm will stay up as long as possible. The property $\llcorner K\tau\lrcorner \rightarrow \llcorner A\lrcorner$ encodes this behavior (Table 6.5):

Theorem 4. *D is maximal for φ in P iff $D \otimes P \models G(\llcorner K\tau\lrcorner \rightarrow \llcorner A\lrcorner)$.*

Table 6.5: *Maximality*

Alarm Condition	**Maximality**
ExactDel(A, β, d)	$G(\llcorner KY^d\beta\lrcorner \rightarrow \llcorner A\lrcorner)$
BoundDel(A, β, d)	$G(\llcorner KO^{\leq d}\beta\lrcorner \rightarrow \llcorner A\lrcorner)$
FiniteDel(A, β)	$G(\llcorner KO\beta\lrcorner \rightarrow \llcorner A\lrcorner)$

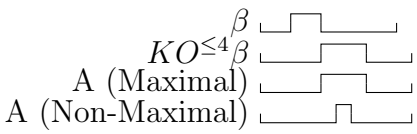

Figure 6.1: Example of Maximal and Non-Maximal traces

Proof. $\Rightarrow$) Suppose D is maximal and by contradiction $D \otimes P \not\models G(\llcorner K\tau\lrcorner \rightarrow \llcorner A\lrcorner)$. Thus, there exists a trace σ_P of P and $i \geq 0$ such that $D(\sigma_P) \times \sigma_P, i \models (\llcorner K\tau\lrcorner \wedge \neg\llcorner A\lrcorner)$ (where $D(\sigma_P)$ is the diagnoser trace matching σ_P as defined in Definition 7). By Definition 10 of $\llcorner\cdot\lrcorner$, i is an observation point. Let i be the j-th observation point of σ_P. Consider D' obtained by $D(\sigma_p)$ converting the trace into a transition system using a sink state so that D' is deterministic and setting $\llcorner A\lrcorner$ to true only in the state $D(\sigma_P)[j]$ (thus triggering A in j and setting it to false at the next observation point). For every trace σ'_P of P matching with $D'(\sigma_P)$, $obs(\sigma'_P) = obs(\sigma_P)$, and thus $\sigma'_P, i \models \tau$ (since $D(\sigma_P) \times \sigma_P, i \models \llcorner K\tau\lrcorner$). Therefore $D' \models G(\llcorner A\lrcorner \rightarrow \tau)$ contradicting the hypothesis.

$\Leftarrow$) Suppose $D \otimes P \models G(\llcorner K\tau\lrcorner \rightarrow \llcorner A\lrcorner)$ and by contradiction D is not maximal for φ in P. Then there exists a trace σ_P of P such that $D(\sigma_P), i \not\models \llcorner A\lrcorner$ and there exists another diagnoser D' of P such that $D'(\sigma_P), i \models \llcorner A\lrcorner$ and $D' \otimes P \models G(\llcorner A\lrcorner \rightarrow \tau)$. Then, for some j, $D(\sigma_P) \otimes \sigma_P, j \not\models \llcorner A\lrcorner$, $D'(\sigma_P) \otimes \sigma_P, j \models \llcorner A\lrcorner$, and so $D(\sigma_P) \otimes \sigma_P, j \not\models \llcorner K\tau\lrcorner$ and $\sigma_P, j \models \tau$. Then there exists another trace σ'_P of P and j' such that $ObsEq((\sigma'_P, j'), (\sigma_P, j))$ and $\sigma'_P, j' \not\models \tau$. Since D' is deterministic, $D'(\sigma'_P)$ and $D'(\sigma_P)$ are equal up

to position i, and so $D' \otimes P \not\models G(\llcorner A \lrcorner \rightarrow \tau)$ contradicting the hypothesis. $\square$

Whenever the diagnoser knows that τ is satisfied, it will raise the alarm. An example of maximal and non-maximal alarm is given in Figure 6.1. Note that according to our definition, the set of maximal alarms is a subset of the non-maximal ones.

The concepts of maximality and trace completeness are related, but different concepts. In maximality, we are not only testing that the diagnoser will raise the alarm, but also that it will do it *maximally* (as early and as long as possible). A diagnoser can be trace complete by raising the alarms in all situations that are diagnosable, but do so with some delay or in a non-maximal way.

Introspection A property related to maximality is the capability of the diagnoser to justify the raising of the alarm. This is a property that has theoretical interest, and it is a stronger version of correctness that shows that the diagnoser is correct and it did not *guess* the raising of the alarm. This property is guaranteed by construction for any correct diagnoser, as shown in the following theorem.

Theorem 5. *Given a diagnoser D and a plant P, for each alarm A of D, with temporal condition τ, if D is correct for A it holds that:*

$$D \otimes P \models G(\llcorner A \lrcorner \rightarrow \llcorner K\tau \lrcorner)$$

Thus, whenever the diagnoser raises an alarm, it knows that the diagnosis condition has occurred.

Proof. We assume by contradiction that the $G(\llcorner A \lrcorner \rightarrow \llcorner K\tau \lrcorner)$ is not satisfied. Therefore, there exist σ and i such that $D(\sigma) \otimes \sigma, i \models \llcorner A \lrcorner \wedge \neg \llcorner K\tau \lrcorner$ (where $D(\sigma_P)$ is the diagnoser trace matching σ_P as defined in Definition 7), which is equivalent to $\llcorner A \lrcorner \wedge \neg K\tau$ (by Definition 10 of $\llcorner \cdot \lrcorner$). Thus, $(\sigma, i) \models \tau$ by

Table 6.6: ASL$_K$ specification patterns among the four dimensions: *Diagnosability*, *Maximality*, *Completeness* and *Correctness*.

	Delay	*Maximality* $= False$	*Maximality* $= True$
Diag $= System$	ExactDel	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge G(\beta \rightarrow X^d \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge G(\beta \rightarrow X^d \llcorner A \lrcorner) \wedge G(\llcorner KY^d \beta \lrcorner \rightarrow \llcorner A \lrcorner)$
	BoundDel	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge G(\beta \rightarrow F^{\leq d} \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge G(\beta \rightarrow F^{\leq d} \llcorner A \lrcorner) \wedge G(\llcorner KO^{\leq d} \beta \lrcorner \rightarrow \llcorner A \lrcorner)$
	FiniteDel	$G(\llcorner A \lrcorner \rightarrow O \beta) \wedge G(\beta \rightarrow F \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O \beta) \wedge G(\beta \rightarrow F \llcorner A \lrcorner) \wedge G(\llcorner KO \beta \lrcorner \rightarrow \llcorner A \lrcorner)$
Diag $= Trace$	ExactDel	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge G(\llcorner (\beta \rightarrow X^d \llcorner KY^d \beta \lrcorner) \rightarrow (\beta \rightarrow X^d \llcorner A \lrcorner) \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge G(\llcorner (\beta \rightarrow X^d \llcorner KY^d \beta \lrcorner) \rightarrow (\beta \rightarrow X^d \llcorner A \lrcorner) \lrcorner) \wedge G(\llcorner KY^d \beta \lrcorner \rightarrow \llcorner A \lrcorner)$
	BoundDel	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge G(\llcorner (\beta \rightarrow F^{\leq d} \llcorner KO^{\leq d} \beta \lrcorner) \rightarrow (\beta \rightarrow F^{\leq d} \llcorner A \lrcorner) \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge G(\llcorner (\beta \rightarrow F^{\leq d} \llcorner KO^{\leq d} \beta \lrcorner) \rightarrow (\beta \rightarrow F^{\leq d} \llcorner A \lrcorner) \lrcorner) \wedge G(\llcorner KO^{\leq d} \beta \lrcorner \rightarrow \llcorner A \lrcorner)$
	FiniteDel	$G(\llcorner A \lrcorner \rightarrow O \beta) \wedge G(\llcorner (\beta \rightarrow F \llcorner KO \beta \lrcorner) \rightarrow (\beta \rightarrow F \llcorner A \lrcorner) \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O \beta) \wedge G(\llcorner (\beta \rightarrow F \llcorner KO \beta \lrcorner) \rightarrow (\beta \rightarrow F \llcorner A \lrcorner) \lrcorner) \wedge G(\llcorner KO \beta \lrcorner \rightarrow \llcorner A \lrcorner)$

correctness of D. In order for the $\neg K\tau$ to hold, we need another trace σ' and j s.t. $ObsEq((\sigma, i), (\sigma', j))$ and $(\sigma', j) \models \neg\tau$. By definition, the diagnoser is deterministic, thus we know that for σ, σ' at points i, j we will have the same value of A. Therefore, $D(\sigma') \otimes (\sigma', j) \models \llcorner A \lrcorner \wedge \neg\tau$ so that D is not correct, thus reaching a contradiction. □

6.3 ASL$_K$ Specifications

The formalization of ASL$_K$ (Table 6.6) is obtained by extending ASL (Table 6.1) with the concepts of maximality, diagnosability and trace completeness. Several simplifications are possible. For example, in the case $Diag = Trace$, we do not always need to verify the completeness due to the following result:

Theorem 6. *Given a diagnoser D for a plant P and a trace diagnosable alarm condition φ, if D is maximal for φ, then D is complete.*

Proof. (ExactDel) For all σ, i if $\sigma, i \models (\beta \rightarrow X^d{}_{\llcorner} KY^d \beta_{\lrcorner})$, then by using the maximality assumption, we know that $\sigma, i \models (\beta \rightarrow X^d{}_{\llcorner} A_{\lrcorner})$; thus, $(\sigma, i) \models (\beta \rightarrow X^d{}_{\llcorner} KY^d \beta_{\lrcorner}) \rightarrow (\beta \rightarrow X^d{}_{\llcorner} A_{\lrcorner})$. Similarly we can prove BoundDel and FiniteDel. □

As a corollary of Theorem 6, the same can be applied also for system diagnosable alarm conditions if P is diagnosable, since system diagnosability implies trace diagnosability:

Theorem 7. *Given an alarm condition for the system diagnosable case, and a diagnoser D for a plant P, if D is maximal for φ and φ is diagnosable in P then D is complete.*

Proof. The theorem follows directly from Theorem 6 and the fact that if D is complete for a trace diagnosable alarm condition that is system diagnosable, then D is also complete for the corresponding system diagnosable alarm condition. □

This Theorem is interesting because it tells us that if a specification that was required to be system diagnosable is indeed system diagnosable, then we can just check whether the diagnoser is maximal and avoid performing the completeness test.

Theorem 8. *For all trace diagnosable and non-maximal* ExactDel *specifications, completeness can be replaced by maximality. Formally, for all σ, $\sigma \models G((\beta \rightarrow X^d{}_{\llcorner} KY^d \beta_{\lrcorner}) \rightarrow (\beta \rightarrow X^d{}_{\llcorner} A_{\lrcorner}))$ iff $\sigma \models G(_{\llcorner} KY^d \beta_{\lrcorner} \rightarrow {}_{\llcorner} A_{\lrcorner})$*

Table 6.7: ASL_K with simplified patterns for $Diag = Trace$

	Template	$Maximality = False$	$Maximality = True$
$Diag = System$	ExactDel	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge G(\beta \rightarrow X^d \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge G(\beta \rightarrow X^d \llcorner A \lrcorner)$ $G(\llcorner KY^d \beta \lrcorner \rightarrow A)$
	BoundDel	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge G(\beta \rightarrow F^{\leq d} \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge G(\beta \rightarrow F^{\leq d} \llcorner A \lrcorner) \wedge$ $G(\llcorner KO^{\leq d} \beta \lrcorner \rightarrow A)$
	FiniteDel	$G(\llcorner A \lrcorner \rightarrow O\beta) \wedge G(\beta \rightarrow F \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O\beta) \wedge G(\beta \rightarrow F \llcorner A \lrcorner) \wedge$ $G(\llcorner KO\beta \lrcorner \rightarrow A)$
$Diag = Trace$	ExactDel	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge$ $G(\llcorner KY^d \beta \lrcorner \rightarrow A)$	$G(\llcorner A \lrcorner \rightarrow Y^d \beta) \wedge$ $G(\llcorner KY^d \beta \lrcorner \rightarrow A)$
	BoundDel	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge$ $G((\beta \wedge F^{\leq d} \llcorner KO^{\leq d} \beta \lrcorner) \rightarrow F^{\leq d} \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O^{\leq d} \beta) \wedge$ $G(\llcorner KO^{\leq d} \beta \lrcorner \rightarrow A)$
	FiniteDel	$G(\llcorner A \lrcorner \rightarrow O\beta) \wedge$ $G((\beta \wedge F \llcorner KO\beta \lrcorner) \rightarrow F \llcorner A \lrcorner)$	$G(\llcorner A \lrcorner \rightarrow O\beta) \wedge$ $G(\llcorner KO\beta \lrcorner \rightarrow A)$

Proof.

$$
\begin{aligned}
\sigma, i \models & ((\beta \rightarrow X^d \llcorner KY^d \beta \lrcorner) \rightarrow (\beta \rightarrow X^d \llcorner A \lrcorner)) && \text{iff} \\
\sigma, i \models & ((\beta \wedge X^d \llcorner KY^d \beta \lrcorner) \rightarrow X^d \llcorner A \lrcorner) && \text{iff} \\
\sigma, i + d \models & ((Y^d \beta \wedge \llcorner KY^d \beta \lrcorner) \rightarrow \llcorner A \lrcorner) && \text{iff} \\
\sigma, i + d \models & ((\llcorner Y^d \beta \wedge KY^d \beta \lrcorner) \rightarrow \llcorner A \lrcorner) && \text{iff} \\
\sigma, i + d \models & (\llcorner KY^d \beta \lrcorner \rightarrow \llcorner A \lrcorner)
\end{aligned}
$$

Therefore, we can conclude that for all i, $\sigma, i \models ((\beta \rightarrow X^d \llcorner KY^d \beta \lrcorner) \rightarrow (\beta \rightarrow X^d \llcorner A \lrcorner))$ iff for all $j \geq d$, $\sigma, j \models (\llcorner KY^d \beta \lrcorner \rightarrow \llcorner A \lrcorner)$. We conclude noting that for $j < d$, $Y^d \beta$ is false and therefore $\sigma, j \models (\llcorner KY^d \beta \lrcorner \rightarrow \llcorner A \lrcorner)$. □

After applying the simplifications specified in Theorem 6 and Theorem 8 and the equivalence $\llcorner \phi \lrcorner \rightarrow \llcorner \psi \lrcorner \equiv \llcorner \phi \lrcorner \rightarrow \psi$, we obtain the table in Table 6.7, where the patterns in the lower half ($Diag = Trace$) have been simplified.

Operational Context To encode the *operational context* in ASL_K, we simply limit the set of traces on which we evaluate the KL_1 properties. For-

mally, a context $\mathcal{C}$ is any LTL formula expressed on the state variables and events of the plant P.

For a specification φ, instead of verifying that $D \otimes P \models \varphi$, we verify that $D \otimes P \models \mathcal{C} \rightarrow \varphi$. This is equivalent to verifying the specification on a restricted plant P' that has the same language as the context, since the context is expressed only on variables of the plant.

6.4 Validation and Verification of ASL$_K$

The formal characterization of ASL$_K$ makes it possible to apply formal methods for the verification and validation of a ASL$_K$ specification.

In *verification*, we check that a candidate diagnoser fulfills a set of requirements.

Definition 21. *Let D be a diagnoser for alarms $\mathcal{A}$ and plant P. We say that D* satisfies a set $\mathcal{A}$ of ASL$_K$ specifications *iff for each φ in $\mathcal{A}$ there exists an alarm $A_\varphi \in \mathcal{A}$ and $D \otimes P \models \varphi$.*

To perform this verification steps, we need a model checker able to deal with the semantics of the plant-FDIR combination (asynchronous/synchronous, bounded/perfect recall, finite/infinite state, etc.). If the specification falls in the pure LTL fragment (ASL) we can verify it with an LTL model-checker, otherwise, we need a model-checker for KL_1. Techniques for model-checking KL_1 will be discussed in Chapter 11.

In *validation* we verify that the requirements capture the interesting behaviors and exclude the spurious ones, before proceeding with the design of the diagnoser. The most typical validation check is diagnosability. We can check diagnosability using temporal epistemic logic (as described in Section 6.2.1) or with other techniques (as we will discuss in Chapter 7). In this section, we provide a more general discussion of validation, in which

we use the temporal epistemic logic characterization of the specification to perform automated reasoning.

Validation

Given a specification $\mathcal{A}$ for our diagnoser, we want to make sure that it captures the designer expectations. Known techniques for requirements validation [56] include checking their *consistency*, and their *realizability*, i.e., whether they can be implemented on a given plant. By construction, an ASL_K specification is always *consistent*, i.e., there cannot be internal contradictions. This is due to the fact that alarm specifications do not interact with each other, and each alarm specification can always be satisfied by a plant. Instead, *realizability* reduces to checking *diagnosability*.

Often we want to show that there exists some condition under which the alarm might be triggered (*possibility*), and some other conditions that require the alarm to be triggered (*necessity*). An alarm that is always (or never) triggered is not useful. Moreover, we might want to identify assumptions on the environment of the diagnoser (including details on the plant) that might have an impact on the the alarms. For example, if we have a single fault assumption for our system, an alarm that implicitly depends on the occurrence of two faults will never be triggered. Similarly, our assumptions on the environment might provide some link between the behavior of different components, or dynamics of faults and thus characterize the relation between different alarms.

The operational context $\mathcal{C}$, expressed as an LTL formula, can be empty, or include detailed information on the behavior of the environment and plant, since throughout the different phases of the development process, we have access to better versions of the plant model, and therefore the analysis can be refined.

Checking *possibility* means checking that the alarms can be eventually

activated, but also that they are not always active. This means that for a given alarm condition $\varphi \in \mathcal{A}$, we are interested in verifying that there are two traces $\sigma, \sigma' \in \mathcal{C}$ s.t. :

$$\sigma \models F\llcorner A_{\varphi} \lrcorner \text{ and } \sigma' \models F\neg\llcorner A_{\varphi} \lrcorner$$

This can be done by checking the (LTL) unsatisfiability of

$$(\mathcal{C} \wedge \varphi) \rightarrow G\neg\llcorner A_{\varphi} \lrcorner \text{ and } (\mathcal{C} \wedge \varphi) \rightarrow G\llcorner A_{\varphi} \lrcorner$$

Checking *necessity* provides us a way to understand whether there is some correlation between alarms. This, in turns, makes it possible to simplify the model, or to guarantee some redundancy requirement. To check whether $A_{\varphi'}$ is a more general alarm than A_{φ} (subsumption) we check whether

$$(\mathcal{C} \wedge \varphi \wedge \varphi') \rightarrow G(\llcorner A_{\varphi} \lrcorner \rightarrow \llcorner A_{\varphi'} \lrcorner)$$

is valid. An example of subsumption of alarms is given by the definition of maximality: any non-maximal alarm subsumes its corresponding maximal version. Finally, we can verify that two alarms are mutually exclusive by checking the validity of $(\mathcal{C} \wedge \varphi \wedge \varphi') \rightarrow G\neg(\llcorner A_{\varphi} \lrcorner \wedge \llcorner A_{\varphi'} \lrcorner)$.

To clarify the concepts presented here, we apply a necessity check on our running example. In the Battery-Sensor, we have two alarms specified on $PSU1$, one ExactDel and one FiniteDel:

- ExactDel$_K(PSU1_{Exact_2}, \beta_{PSU1}, 2, Trace, True)$
- BoundDel$_K(PSU1_{Bound}, \beta_{PSU1}, 2, Trace, True)$

Our goal is to show that $PSU1_{Exact_i}$ is more specific than (is subsumed by) $PSU1_{Bound}$. This means that for any plant and diagnoser, the following holds:

$$D \otimes P \models (\varphi_{PSU1_{Exact_2}} \wedge \varphi'_{PSU1_{Bound}}) \rightarrow G(\llcorner PSU1_{Exact_2} \lrcorner \rightarrow \llcorner PSU1_{Bound} \lrcorner)$$

Since we want to show this for any plant and diagnoser, this reduces to checking the validity of

$$(\varphi_{PSU1_{Exact_2}} \wedge \varphi_{PSU1_{Bound}}) \rightarrow G(\llcorner PSU1_{Exact_2} \lrcorner \rightarrow \llcorner PSU1_{Bound} \lrcorner)$$

By looking at Table 6.7 we can see the definition of both alarms:

- $G(\llcorner PSU1_{Exact_i} \lrcorner \rightarrow Y^i \beta_{PSU1}) \wedge G(\llcorner KY^i \beta_{PSU1} \lrcorner \rightarrow \llcorner PSU1_{Exact_i} \lrcorner)$
- $G(\llcorner PSU1_{Bound} \lrcorner \rightarrow O^{\leq C} \beta_{PSU1}) \wedge G(\llcorner KO^{\leq C} \beta_{PSU1} \lrcorner \rightarrow \llcorner PSU1_{Bound} \lrcorner)$

By renaming with $PE = PSU1_{Exact_2}$ and $PB = PSU_{Bound}$ (for brevity) and expanding the definitions of $\varphi_{PSU1_{Exact_2}} \wedge \varphi_{PSU1_{Bound}}$ we have that

$$\begin{aligned}(G(\llcorner PE \lrcorner \rightarrow Y^2 \beta) \wedge G(\llcorner KY^2 \beta \lrcorner \rightarrow \llcorner PE \lrcorner) \wedge \\ G(\llcorner PB \lrcorner \rightarrow O^{\leq 2} \beta) \wedge G(\llcorner KO^{\leq 2} \beta \lrcorner \rightarrow \llcorner PB \lrcorner)) \\ \rightarrow G(\llcorner PE \lrcorner \rightarrow \llcorner PB \lrcorner)\end{aligned}$$

We can apply Theorem 5 (Introspection of the Diagnoser), and therefore write:

$$\begin{aligned}(G(\llcorner PE \lrcorner \rightarrow Y^2 \beta) \wedge G(\llcorner KY^2 \beta \lrcorner \rightarrow \llcorner PE \lrcorner) \wedge \\ G(\llcorner PB \lrcorner \rightarrow O^{\leq 2} \beta) \wedge G(\llcorner KO^{\leq 2} \beta \lrcorner \rightarrow \llcorner PB \lrcorner) \wedge \\ G(\llcorner PE \lrcorner \rightarrow \llcorner KY^2 \beta \lrcorner) \wedge G(\llcorner PB \lrcorner \rightarrow \llcorner KO^{\leq 2} \beta \lrcorner)) \\ \rightarrow G(\llcorner PE \lrcorner \rightarrow \llcorner PB \lrcorner)\end{aligned}$$

To prove that the above formula is valid we prove that its negation is *unsatisfiable*:

$$\begin{aligned}(G(\llcorner PE \lrcorner \rightarrow Y^2 \beta) \wedge G(\llcorner KY^2 \beta \lrcorner \rightarrow \llcorner PE \lrcorner) \wedge \\ G(\llcorner PB \lrcorner \rightarrow O^{\leq 2} \beta) \wedge G(\llcorner KO^{\leq 2} \beta \lrcorner \rightarrow \llcorner PB \lrcorner) \wedge \\ G(\llcorner PE \lrcorner \rightarrow \llcorner KY^2 \beta \lrcorner) \wedge G(\llcorner PB \lrcorner \rightarrow \llcorner KO^{\leq 2} \beta \lrcorner)) \\ \wedge \neg G(\llcorner PE \lrcorner \rightarrow \llcorner PB \lrcorner)\end{aligned}$$

The first part of this formula is composed by conjuncts in the form $G\psi$. This means that a counter examples is a trace for which each state satisfies ψ. Moreover, we need one of these states to satisfy $(PE \wedge \neg\llcorner PB\lrcorner)$. Therefore, to prove the unsatisfiable of the above formula, we can just prove that no state exists that satisfies:

$$\begin{aligned}(\llcorner PE\lrcorner \rightarrow Y^2\beta) \wedge (\llcorner KY^2\beta\lrcorner \rightarrow \llcorner PE\lrcorner) \wedge \\ (\llcorner PB\lrcorner \rightarrow O^{\leq 2}\beta) \wedge (\llcorner KO^{\leq 2}\beta\lrcorner \rightarrow \llcorner PB\lrcorner) \wedge \\ (\llcorner PE\lrcorner \rightarrow \llcorner KY^2\beta\lrcorner) \wedge (\llcorner PB\lrcorner \rightarrow \llcorner KO^{\leq 2}\beta\lrcorner)) \\ \wedge\llcorner PE\lrcorner \wedge \neg\llcorner PB\lrcorner\end{aligned}$$

We show this by a contradiction since:

$$\begin{aligned}
& \cdots \wedge \llcorner PE\lrcorner \wedge \neg\llcorner PB\lrcorner \\
\textbf{\textit{ObsPoint}}\ \textbf{Def.}\quad & \cdots \wedge \llcorner\top\lrcorner \wedge PE \wedge \neg PB \\
\textbf{Theorem 5 on PE}\quad & \cdots \wedge \llcorner\top\lrcorner \wedge PE \wedge \neg PB \wedge KYY\beta \\
\textbf{Maximality of PB}\quad & \cdots \wedge \llcorner\top\lrcorner \wedge PE \wedge \neg PB \wedge KYY\beta \wedge \neg KO^{\leq 2}\beta \\
\textbf{†Def. of } \neg K\quad & \cdots \wedge \llcorner\top\lrcorner \wedge PE \wedge \neg PB \wedge KYY\beta \wedge \neg O^{\leq 2}\beta \\
\textbf{Def. of } O^{\leq n}\quad & \cdots \wedge \llcorner\top\lrcorner \wedge PE \wedge \neg PB \wedge KYY\beta \wedge \neg(\beta \vee Y\beta \vee YY\beta) \\
\textbf{K Axiom } (K\phi \rightarrow \phi)\quad & \cdots \wedge \llcorner\top\lrcorner \wedge PE \wedge \neg PB \wedge YY\beta \wedge \neg\beta \wedge \neg Y\beta \wedge \neg YY\beta
\end{aligned}$$

Thus reaching a contradiction between $YY\beta$ and $\neg YY\beta$. In the step marked with † we need to show that two observationally equivalent traces exists s.t. one satisfies $O^{\leq 2}\beta$ and the other $\neg O^{\leq 2}\beta$; therefore, we only need to show that one of the two (namely $\neg O^{\leq 2}\beta$) does not exist. □

This example shows that by defining a semantics for ASL_K based on temporal epistemic logic, we can apply automated satisfiability and model-checking techniques to prove properties of interest on a given alarm specification.

6.5 Examples

Battery Sensor System

An ASL_K specification is built by instantiating the patterns defined in Table 6.6. For example, we write $\textsc{ExactDel}_K(A, \beta, d, Trace, True)$ for an exact-delay alarm A for β with delay d, that satisfies the trace diagnosability property and is maximal.

$$\begin{array}{|l|}\hline \textsc{ExactDel}_K(PSU1_{Exact_i}, \beta_{PSU1}, i, Trace, True) \\ \textsc{BoundDel}_K(PSU1_{Bound}, \beta_{PSU1}, C, Trace, True) \\ \textsc{BoundDel}_K(BS, \beta_{BS}, DC, Trace, True) \\ \textsc{FiniteDel}_K(Discharged, \beta_{Depleted}, Trace, False) \\ \textsc{FiniteDel}_K(B1Leak, \beta_{Battery1}, System, True) \\ \hline \end{array}$$

Figure 6.2: ASL_K Specification for the BSS

Figure 6.2 shows how we extend the specification for the Battery Sensor System by introducing requirements on the diagnosability and maximality of alarms. In particular, all the alarms that we defined are not system diagnosable. Therefore, we need to weaken the requirements and make them trace-diagnosable. The patterns are then converted into temporal epistemic formulas as shown in Figure 6.3.

Alarm	**Formula**
$PSU1_{Exact_i}$	$G(\llcorner PSU1_{Exact_i} \rightarrow Y^i \beta_{PSU1}) \wedge G(\llcorner KY^i \beta_{PSU1} \rightarrow {}\llcorner PSU1_{Exact_i})$
$PSU1_{Bound}$	$G(\llcorner PSU1_{Bound} \rightarrow O^{\leq C} \beta_{PSU1}) \wedge G(\llcorner KO^{\leq C} \beta_{PSU1} \rightarrow {}\llcorner PSU1_{Bound})$
BS	$G(\llcorner BS \rightarrow O^{\leq DC} \beta_{BS}) \wedge G(\llcorner KO^{\leq DC} \beta_{BS} \rightarrow {}\llcorner BS)$
$Discharged$	$G(\llcorner Discharged \rightarrow O\beta_{Deplated}) \wedge G((\beta_{Deplated} \wedge F\llcorner KO\beta_{Deplated}) \rightarrow F\llcorner Discharged)$
$B1Leak$	$G(\llcorner B1Leak \rightarrow O\beta_{Battery1}) \wedge G(\beta_{Battery1} \rightarrow F\llcorner B1Leak) \wedge G(\llcorner KO\beta_{Battery1} \rightarrow {}\llcorner B1Leak)$

Figure 6.3: KL_1 translation of ASL_K patterns for the BSS

The Battery Leak is trace diagnosable but not system diagnosable. This means that in general, we cannot detect the battery leak, but there is at least one execution in which we can. In particular, this is the execution in

which the mode becomes Secondary 2 when Battery 1 was charged, and we can see the battery discharging, thus detecting the fault. Note that to detect this fault, we need to recall the fact that previously the battery was charged, and therefore a diagnoser with 0 recall would not be able to detect this fault. To study the other faults, we setup an operational context. In particular, we assume that at most one fault can occur. Under this assumption, the sensor faults are trace diagnosable, since there is an execution in which the device stops working because of a discrepancy in sensor readings.

Lets assume that we are given a diagnoser D for our system, and we want to verify that it satisfies the alarm specification, e.g., $PSU1_{Exact}$ under the context of a single fault:

$$D \otimes P \models G(count(faults) \leq 1) \rightarrow [\\ G(\llcorner PSU1_{Exact_i} \lrcorner \rightarrow Y^i \beta_{PSU1}) \wedge G(\llcorner KY^i \beta_{PSU1} \lrcorner \rightarrow \llcorner PSU1_{Exact_i} \lrcorner)]$$

where $count(faults) \leq 1$ encodes the possibility for at most one fault to occur at any given time[1]. We can also split the verification in two separate verification tasks: correctness and maximality. In this way, *correctness* can be verified using a simple LTL model-checker rather than an epistemic one:

$$D \otimes P \models G(count(faults) \leq 1) \rightarrow G(\llcorner PSU1_{Exact_i} \lrcorner \rightarrow Y^i \beta_{PSU1})$$

Magicbox

A magicbox [90] is a grid-like structure, in which a ball is able to jump from one cell to another according to a predefined pattern. The movement of the ball is not observable directly, but only through two types of observation points: row and columns. An observer on a row is able to understand when the ball is in its row. However, the observer cannot say anything about the

[1]This encoding is weak, in the sense that multiple occurrence of different transient faults are allowed.

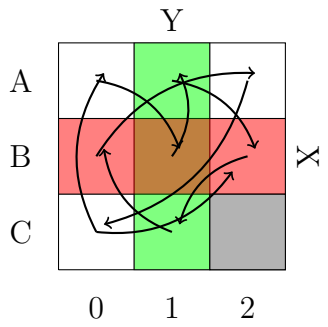

Figure 6.4: A simple magicbox.

distance of the ball, therefore no information on the column can be deduced by this observer. Similarly, the column observers can tell when the ball is in the given column, but nothing more. Our running example is the magicbox in Figure 6.4. This is a 3x3 magicbox, with two observers. The observer x is able to tell when the ball is in the row B, while the observer y is able to tell when the ball is in the column 1. In our asynchronous setting, this means that the events $X = B$ and $Y = 1$ are observable. We also assume that we have a starting state that is outside of the board, and that the first transition places the ball in one of the possible cells, thus giving us an initial observation. The trajectory of the ball is represented by the arrows, e.g., from `A0` the ball will go to `B1`. To keep the example simple, we blocked the cell `C2`.

We want to develop a diagnoser that is able to detect the passage of the ball through a certain cell, even if the cell is not observable by itself. Although the ball path is predefined, there is non-determinism in the initial location of the ball and in the possible non-deterministic transitions. For example, from `C0` the ball can either go to `A0` or to `B2`.

We will come back to this toy example later on, since this provides a simple way of generating scalable benchmarks of partially observable systems. We can change the size of the grid, the number of transitions, the number of non-deterministic transitions, constraint the initial position of the ball, and change the observable rows and columns. Moreover, we

can define several diagnosis and alarm conditions. The simplest diagnosis conditions are by cell, e.g., $\beta_{C1} = \texttt{C1}$. This can be compared to fault identification, in which we are interested in knowing exactly where the ball is. However, we might loose specificity and say that we want to know if the ball is in one of a given set of cells, e.g., the column A: $\beta_A = \texttt{A0} \vee \texttt{A1} \vee \texttt{A2}$ (comparable to fault isolation).

In ASL_K it is also possible to define temporal properties as diagnosis condition. There are two ways of getting into B2, therefore, we might be interested in knowing whether we arrived from C0 rather than from A1; this can be expressed with the temporal condition $\beta_{C0B2} = \texttt{B2} \wedge Y(\texttt{C0})$. This shows how to define informative alarms.

Our example specification consists of the following alarms:

$$\begin{aligned}\mathcal{A} = \{ & \\ & \varphi_1 = \textsc{ExactDel}_K(A(B1), \beta_{B1}, 0, system, True), \\ & \varphi_2 = \textsc{ExactDel}_K(A(C1), \beta_{C1}, 0, trace, True), \\ & \varphi_3 = \textsc{BoundDel}_K(A(B0), \beta_{B0}, 2, system, True) \\ \}&\end{aligned}$$

and we assume the diagnoser to have perfect recall. We now want to see whether the alarm specifications are diagnosable (system or trace depending on the requirement).

For φ_1 we need to check for system diagnosability: $P \models G(\beta_{B1} \rightarrow \llcorner K\beta_{B1} \lrcorner)$. We obtain a counter-example in the initial state: we cannot distinguish whether the ball is starting in B0 or B2. Therefore, there is no diagnoser that is able to satisfy the specification φ_1. This is a simple example that shows why system diagnosability is often difficult to satisfy.

For φ_2, we relax the diagnosability condition, and we only require trace diagnosability. Notice, that this is necessary, since we have the same problem of uncertainty of the initial state between C1 and A1. To understand

for which traces this specification is (trace) diagnosable, we test whether it is non-diagnosable: i.e., $P \models G(\beta_{C1} \rightarrow \neg\llcorner K\beta_{C1}\lrcorner)$. The counter-example to this property is the finite trace $(\texttt{B0}, \texttt{C1})$; therefore, the specification is trace diagnosable. In particular, notice that the only other cell with the same observation as `C1` is `A1` (i.e., $\neg X \wedge Y$). `C1` can be reached only from `B2` that has observation $X \wedge \neg Y$, while `A1` is reachable only from `B1` that has observation $X \wedge Y$. What this means is that we can always know whether the ball is in `C1` as long as we have (and recall) the previous observation. A diagnoser with recall 1 would be sufficient. We can validate this intuition, by using an operational context to exclude executions that start in `C1`, and run a system diagnosability test: $P \models \neg C1 \rightarrow G(\beta_{C1} \rightarrow \llcorner K\beta_{C1}\lrcorner)$

Finally, we consider φ_3. This specification is system diagnosable, since we are allowed a delay of 2. Initially, we cannot distinguish starting in `B0` or `B2`. However, on the next step, from `B0` we can only reach `A2` (with observation $\neg X \wedge \neg Y$), while from `B2` we can only reach `C1` (with observation $\neg X \wedge Y$). The question, therefore becomes whether 2 is the minimum necessary delay for system diagnosability, or whether we can strengthen it to 1. To check this, we run the following system diagnosability query: $P \models G(\beta_{B0} \rightarrow F^{\leq 1}\llcorner KO^{\leq 1}\beta_{B0}\lrcorner)$. Since we obtain a positive answer, we can strengthen our alarm specification and write:

$$\varphi_3' = \textsc{BoundDel}_K(A(B0), \beta_{B0}, 1, system, True)$$

6.6 Chapter Summary

In this chapter, we introduced ASL_K, and its temporal epistemic logic characterization. In Table 6.7 we summarized how the different alarm specification requirements can be captured in ASL_K. We discussed the validation process that temporal epistemic logic characterization enables. Finally, we define and discuss several specifications for both the Battery

Sensor System, and the magicboxes.

Chapter 7

Diagnosability

Faults might not be detectable in a system. This might be due to a limited number of observables, limited recall or by the nature of the system. Given an alarm specification, it is important to verify whether there exists a diagnoser that can satisfy it. This information is crucial during the design phase, because it allows us to refine the specification early in the process.

The problem of whether a fault can be diagnosed (i.e., detected and identified) is called *diagnosability*. We provided a formal characterization of diagnosability in Section 5.4, and in Section 6.2.1 we discussed ways of encoding the problem into temporal epistemic logic. In this chapter, instead, we are interested in extending the problem of diagnosability to a synthesis problem. The problem of *observability synthesis* (sometimes called *sensor placement*) is the problem of optimizing the set of sensors used while preserving (system) diagnosability.

In order to address this problem, we present a different technique that is used to verify *system diagnosability*: the *twin-plant* approach. This technique allows us to reduce the diagnosability test into an LTL model-checking problem, making it easier to map the synthesis problem into a parameter synthesis problem.

In Section 7.1 we extend the twin-plant construction in order to deal

with different types of recall, and explain how to apply this construction to the verification of alarm specifications. In Section 7.2 we use the twin-plant construction to solve the problem of sensor placement. In particular, we show how the problem can be reduced to a monotonic parameter synthesis problem. By associating cost functions to the sensors, we can perform multi-objective optimization and compute the Pareto optimal solutions. An effective technique for solving monotonic parameter synthesis problems is presented and evaluated.

The contributions of this Chapter are:

1. The mapping between alarm specifications (ASL) and twin-plant construction;
2. The extension of the twin-plant construction to deal with bounded recall;
3. An algorithm for computing the Pareto-optimal solutions of a monotonic parameter synthesis problem (originally presented in [23]).

7.1 Verification via Twin-Plant

The twin plant approach was introduced in [101], and it is based on the idea that disproving diagnosability requires finding a *critical pair*. In turn, a critical pair can be seen as the execution of two copies of the plant that have the same observable behavior. A critical pair occurs when one of the two copies is in a state satisfying the diagnosis condition, while the other is not. By forcing the observations to coincide, we perform pruning of the search state, and let the model-checker explore only the pairs of traces that share the same observations. The key advantage of this technique is the possibility of using standard LTL model-checking tools [55] to perform the diagnosability test. The cost of the technique is that we double the size of

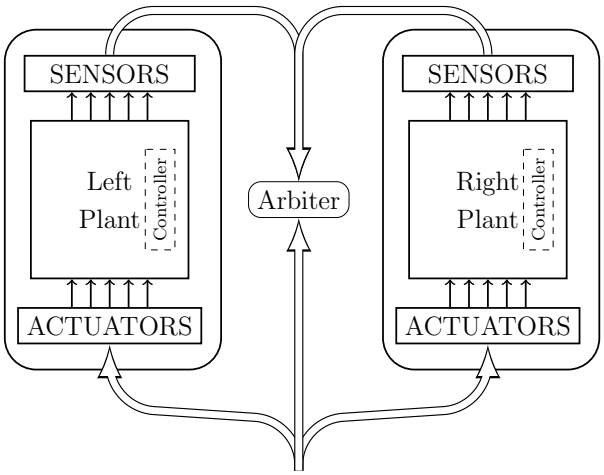

Figure 7.1: Twin-Plant Schema

the model (in the number of variables needed). Therefore, the twin-plant might become too big to handle for the model-checker.

In this Chapter, we use the framework of synchronous transition systems, as done in [55, 26]. This makes it more intuitive to define concepts such as bounded delay. In general, however, the composition of the twin-plant must respect the composition semantics that will be used between the plant and diagnoser: accounting for synchronicity, recall and observability. The results can be extended to the asynchronous case, but this is left as future work.

7.1.1 Twin-Plant Construction

Given a partially observable plant P, the *coupled twin plant* (twin plant) is obtained from P by making the synchronous product of two copies of the plant (P_L and P_R) over the observable events:

Definition 22 (Twin-Plant)**.** *Let $P = \langle V, V_o, T, I\rangle$ be a partially observable transition system. Let $P_L = \langle V^L, V_O^L, T^L, I^L\rangle$ and $P_R = \langle V^R, V_O^R, T^R, I^R\rangle$ be two copies of P s.t. all variables have been renamed. The* twin-plant P^2

is obtained by synchronous product of P_L and P_R:

$$P^2 = P_L \times P_R$$

We refer to P_L and P_R as the *left* and *right* plant as shown in Figure 7.1. In the composition above, the two plants are unrelated. We *couple* them through their observations by adding an arbiter component that raises a flag whenever the two plants have matching observations. The arbiter $S_{obs\_eq} = \langle \{obs\_eq\}, V_O^L \cup V_O^R, T, I \rangle$, where T and I define the invariant:

$$obs\_eq \leftrightarrow \bigwedge_{o \in V_O} o_L = o_R$$

Definition 23 (Coupled Twin-Plant)**.** *The* coupled twin-plant *is a twin-plant with an additional arbiter component:*

$$P^2 \times S_{obs\_eq} = P_L \times S_{obs\_eq} \times P_R$$

The traces in which the observations from both plants match (i.e., *obs_eq* is set to true) are candidates for being critical pairs.

A simple way of thinking about the twin-plant construction is to think about the left and right plant as having different roles. One of them, e.g. the left plant, simulates the *actual* state of the plant during an execution. The other, e.g. the right plant, simulates a possible estimation done by the diagnoser and compatible with the observations. A trace σ of P^2 can be decomposed in the two traces $\langle \sigma_L, \sigma_R \rangle$ of P_L and P_R respectively. Every trace will thus give us a concrete execution of the plant and an educated guess of a diagnoser. Intuitively, a plant is diagnosable for a given condition (e.g., a fault) if for every trace σ of P^2 in which *obs_eq* holds, if σ_L contains the fault also σ_R contains the fault. If this is not the case, then we have found a *critical pair*, in particular $\langle \sigma_L, \sigma_R \rangle$ is a critical pair, since both σ_L and σ_R have the same observable behavior, but disagree on the occurrence of the fault. As discussed in Section 5.2, we are interested in bounding the

delay of the detection and thus we do not require that the two traces have the same observable behavior forever.

7.1.2 Alarm Conditions Verification

To verify system diagnosability using the twin-plant, we need to create a model-checking query that captures our target alarm condition. We associate an LTL property to each alarm condition.

To account for the delay in diagnosis, a critical pair does not require that the two traces are observationally equivalent forever, but only for a certain amount of time after the occurrence of β. Table 7.1 summarizes the LTL properties associated with checking diagnosability using the twin-plant for different types of alarms. A critical pair is a trace of the coupled

Table 7.1: Twin-Plant Diagnosability – Perfect Recall

Exact Delay	$G(\beta_L \rightarrow X^d(H(obs\_eq) \rightarrow Y^d\beta_R)$
Bounded Delay	$G(\beta_L \rightarrow F^{\leq d}(H(obs\_eq) \rightarrow O^{\leq d}\beta_R))$
Finite Delay	$G(\beta_L \rightarrow F(H(obs\_eq) \rightarrow O\beta_R))$

twin-plant in which *obs_eq* holds for sufficiently long after the occurrence of the condition on one of the twins, while the condition does not occur in the other twin. A twin-plant has a critical pair if there is a trace that violates one of the properties of Table 7.1:

Exact Delay $\sigma \models F(\beta_L \wedge X^d(H(obs\_eq) \wedge Y^d\neg\beta_R)$: β occurs on the left plant, both traces are still observationally equivalent after d steps, but the right plant does not satisfy β at the same time of the left plant. Since the X and Y cancel each other out, we could move the $\neg\beta_R$ out, obtaining:

$$F(\beta_L \wedge \neg\beta_R \wedge X^d(H(obs\_eq))$$

but we maintain the original formulation for symmetry with the other delays.

Bounded Delay $\sigma \models F(\beta_L \wedge G^{\leq d}(H(obs\_eq) \wedge H^{\leq d}\neg\beta_R))$

β occurs on the left plant and for any future point within the delay, the two traces are observationally equivalent. However, it is never the case that β_R holds within the delay.

Finite Delay $\sigma \models F(\beta_L \wedge G(H(obs\_eq) \wedge H\neg\beta_R))$

we have an occurrence of the diagnosis condition β on the left plant (β_L) and the whole trace σ is *obs_eq*. At the same time, on the right plant β never holds. This can be simplified as:

$$F(\beta_L \wedge G(H(obs\_eq \wedge \neg\beta_R)))$$

Using the result from Theorem 1, we can rewrite Sampath's diagnosability test for a given delay d as:

$$P^2 \times S_{obs\_eq} \models G(\beta_L \rightarrow F^{\leq d}(H(obs\_eq) \rightarrow O\beta_R))$$

7.1.3 Bounded Recall

The classical twin-plant construction [101, 137] enforces perfect recall, meaning that the observations on the plants are synchronized from the beginning of time, and there is no way to forget what has happened. We capture this in the verification properties (by imposing $H(obs\_eq)$) we require that the traces are observationally equivalent from the beginning of the execution. This can be extended to enable the encoding of the properties for other types of recall. In particular, we consider *clock semantics* [98] as a stronger version of bounded recall. In the clock semantics, in addition to the bounded recall, both plants have access to a *global clock*. This provides an additional source of information, thus enabling a finer-grain estimation of the state of the plant. In order of expressiveness, we have bounded recall, clock semantics, and perfect recall.

The diagnosability properties for clock semantics, are presented in Table 7.2, where $\mathcal{R} \in [0, \infty]$, indicates the recall. By assuming that $H^{\leq\infty} \equiv H$ we obtain a unified definition for perfect recall and clock semantics.

Table 7.2: Twin-Plant Diagnosability – Clock Semantics

Finite Delay	$G(\beta_L \rightarrow X^d(H^{\leq\mathcal{R}}(obs\_eq) \rightarrow Y^d\beta_R)$
Bounded Delay	$G(\beta_L \rightarrow F^{\leq d}(H^{\leq\mathcal{R}}(obs\_eq) \rightarrow O^{\leq d}\beta_R))$
Finite Delay	$G(\beta_L \rightarrow F(H^{\leq\mathcal{R}}(obs\_eq) \rightarrow O\beta_R))$

The clock-semantics discussed here comes from the fact that the plants are in a synchronous composition. Even if there is no variable sharing, both systems will evolve at the same speed. In order to talk about bounded recall, we need to loose the synchronicity of the two plants.

To model the lack of memory of the plants, we would like them to be able to start from any of their reachable states. This would remove the synchronicity between the two systems. However, computing the reachable state set is usually unfeasible or impossible (e.g., for infinite state systems). Therefore, we encode this behavior by letting the plants evolve freely for a non-deterministic amount of time, before starting the synchronization of both plants. We add a non-deterministic flag *loop* to the plants. As long as *loop* is true, the plant will remain (stutter) in the initial state. Once *loop* becomes false, it will stay false, and the plant will start evolving according to the transition relation.

Definition 24. *(P_{loop}) Given a plant P, we obtain $P_{loop} = \langle V^P \cup \{loop\}, V_O^P \cup \{loop\}, T^{loop}, I^P \rangle$, where:*

$$T^{loop} \triangleq (loop \rightarrow \bigwedge_{v \in V^P} v' = v) \wedge (\neg loop \rightarrow \neg loop')$$

When talking about the twin-plant, we define *loop* as $loop_L \vee loop_R$ i.e., at least one of the plants is looping. Properties described in Table 7.3 will

be considered only when both plants are not looping anymore:

$$G\varphi \text{ becomes } G(\neg loop \rightarrow \varphi)$$

Moreover, we add $loop_L, loop_R$ to the observable set. Intuitively, this describes the fact that in a bounded recall system, we know how many observations we did so far. For example, we cannot consider a critical pair in which the left plant has performed 3 observations and the right plant has performed 5. This is normally captured by the semantics of H, but is lost when we introduce the initial loops, therefore, we need to consider $loop$ as observable.

Table 7.3: Twin-Plant Diagnosability – Bounded Recall

Exact Delay	$G(\neg loop \wedge \beta_L \rightarrow X^d(H^{\leq \mathcal{R}}(obs\_eq) \rightarrow Y^d \beta_R)$
Bounded Delay	$G(\neg loop \wedge \beta_L \rightarrow F^{\leq d}(H^{\leq \mathcal{R}}(obs\_eq) \rightarrow O^{\leq d} \beta_R))$
Finite Delay	$G(\neg loop \wedge \beta_L \rightarrow F(H^{\leq \mathcal{R}}(obs\_eq) \rightarrow O\beta_R))$

The formulation presented in Table 7.3 can be used for perfect recall, clock semantics and bounded recall, by allowing the systems to perform initial loop (bounded recall) or not (perfect recall and clock semantics), and by considering $H^\infty \equiv H$. By comparing Table 7.3 with Table 6.2 we can see that the epistemic formulation and the LTL formulation follow the same structure, with the constraint on the observability and recall being captured by the K operator.

7.1.4 Formal Relation

The coupled twin-plant and LTL encodings from Table 7.3 capture the definitions of diagnosability. First, we adapt the definitions of diagnosability (Definitions 13, 14 and 16) to the synchronous context, in which every point is an observation point.

Definition 25 (Exact Delay (Synchronous Case))**.** *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$ and a set of observables O, we say that* ExactDel(A, β, d) *is* system diagnosable *in P iff for all σ_1, i s.t. $\sigma_1, i \models \beta$ and for all σ_2, j, if $ObsEq_O^{\mathcal{R}}((\sigma_1, i+d), (\sigma_2, j+d))$, then $\sigma_2, j \models \beta$.*

Theorem 9. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, and a set of observables O, we say that* ExactDel(A, β, d) *is* system diagnosable *in P iff*

$\mathcal{R} \neq \infty$) $P^2_{loop} \times S_{obs\_eq} \models G(\neg loop \wedge \beta_L \rightarrow X^d(H^{\leq \mathcal{R}}(obs\_eq) \rightarrow Y^d \beta_R)$

$\mathcal{R} = \infty$) $P^2 \times S_{obs\_eq} \models G(\beta_L \rightarrow X^d(H(obs\_eq) \rightarrow Y^d \beta_R)$

i.e., the coupled twin-plant constructed from P satisfies the Exact Delay property from Table 7.1 (for perfect recall) or Table 7.3 for (bounded recall).

Proof. $\Rightarrow$ (By contradiction) We assume that there is a trace of the twin-plant that violates the property. This trace can be decomposed in two traces σ_L,σ_R, and we know that there is a point i s.t. $\sigma_L, i \models \beta_L$, $\sigma_R, i \models \neg\beta_R$, and that $ObsEq_O^{\mathcal{R}}((\sigma_L, i+d), (\sigma_R, i+d))$. Thus, we reach a contradiction. This direction of the proof works both for perfect and bounded recall, with the minor change that for bounded recall, we need to require that $\beta_L, i \models \neg loop$, in order to be able to map the critical pair into a real execution of the system, and not in an artifact of the initial stuttering.

$\Leftarrow$ (By contradiction) We assume that there is a pair of pointed traces (σ_1, i) and (σ_2, j) s.t. $ObsEq_O^{\mathcal{R}}((\sigma_1, i+d), (\sigma_2, j+d))$ and $\sigma_1, i \models \beta$ but $\sigma_2, j \models \neg\beta$. We need to show that the two traces belong to the twin-plant and cannot satisfy the LTL property, to do so, we distinguish between perfect and bounded recall.

Perfect Recall Due to the perfect recall and synchronicity assumptions, both traces must have the same length to be observationally equivalent. Therefore, we deduce that $i = j$, and can see that the trace representing (σ_1, σ_2) in the twin-plant violates the LTL property.

Bounded Recall Due to the bounded recall, we cannot say anything on the relative value of i and j[1]. Intuitively, to convert these traces into traces of the twin-plant, we need to enforce the two traces to have the same length. If $j = i$ we are done, so we assume that $j < i$ (the other case is symmetric). We prefix σ_2 with σ_{loop}^{i-j} that is a finite trace obtained by stuttering in the initial state $i-j$ times. The twin-plant trace is obtained as $(\sigma_1, \sigma_{loop}^{i-j}\sigma_2)$, that violates the LTL property, since $\sigma, i \models \beta_L \wedge \neg\beta_R$. Note that the recalled fragments have the same length, i.e., $|obs_O^{\mathcal{R}}(\sigma_1, i)| = |obs_O^{\mathcal{R}}(\sigma_2, j)|$ otherwise the two traces would not be observationally equivalent to begin with.

□

Definition 26 (Bounded Delay (Synchronous Case))**.** *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, and a set of observables variables O, we say that* BoundDel(A, β, d) *is* system diagnosable *in P iff forall σ_1, i s.t. $\sigma_1, i \models \beta$ there exists k s.t. $i \leq k \leq i + d$, and for all σ_2, l, if $ObsEq_O^{\mathcal{R}}((\sigma_1, k), (\sigma_2, l))$, then there exists j s.t. $l - d \leq j \leq l$ and $\sigma_2, j \models \beta$.*

Theorem 10. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, and a set of observables O, we say that* BoundDel(A, β, d) *is* system diagnosable *in P iff*

$\mathcal{R} \neq \infty$) $P_{loop}^2 \times S_{obs\_eq} \models G(\neg loop \wedge \beta_L \rightarrow F^{\leq d}(H^{\leq \mathcal{R}}(obs\_eq) \rightarrow O^{\leq d}\beta_R))$

[1]In clock semantics we could make the same assumption as for perfect recall

$\mathcal{R} = \infty$) $P^2 \times S_{obs\_eq} \models G(\beta_L \rightarrow F^{\leq d}(H(obs\_eq) \rightarrow O^{\leq d}\beta_R))$

i.e., the coupled twin-plant constructed from P satisfies the Bounded Delay property from Table 7.1 (for perfect recall) or Table 7.3 (for bounded recall).

Proof. $\Rightarrow$ (By contradiction) We assume that there is a trace of the twin-plant that violates the property. This trace can be decomposed in two traces σ_L,σ_R, s.t. there is a point i in which $\sigma_L, i \models \beta_L$, a point k s.t. $ObsEq_O^{\mathcal{R}}((\sigma_L, k), (\sigma_R, k))$ and $i \leq k \leq i + d$ and a point $i - d \leq j \leq k$ s.t. $\sigma_R, j \models \neg\beta_R$. Thus we reach a contradiction.

$\Leftarrow$ (By contradiction) We assume that there is a a pair of traces and points (σ_1, i), (σ_2, l), and forall points k $(i \leq k \leq i + d)$ s.t. $\sigma_1, i \models \beta$, $ObsEq_O^{\mathcal{R}}((\sigma_1, k), (\sigma_2, l))$ and for all j $(l - d \leq j \leq l)$ $\sigma_2, j \models \neg\beta$.

Perfect Recall As done in the previous case, using the assumption of synchronicity and perfect recall, we know that $k = l$. Therefore, the trace of the twin plant is obtained by simply composing σ_1 and σ_2. Note that the index k indicates the point that satisfies the $F^{\leq d}$ and j is the point that satisfies $O^{\leq d}$.

Bounded Recall We cannot assume anything about the relative position of k and l. Therefore, we introduce stuttering in the initial state until $k = l$. The twin-plant trace is obtained (for $l < k$) as $(\sigma_1, \sigma_{loop}^{k-l}\sigma_2)$. This trace does not satisfy the LTL specification, thus reaching a contradiction.

□

Definition 27 (Finite Delay (Synchronous Case))**.** *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, and a set of observables O, we say that* FiniteDel(A, β) *is* system diagnosable *in P iff for all σ_1, i s.t. $\sigma_1, i \models \beta$ then there exist $k \geq i$ s.t. for all σ_2, l if $ObsEq_O^{\mathcal{R}}((\sigma_1, k), (\sigma_2, l))$ then there exists $j \leq l$ $\sigma_2, j \models \beta$.*

Theorem 11. *Given a plant P, a diagnosis condition β, a recall $\mathcal{R}$, and a set of observables O, we say that* FiniteDel(A, β) *is* system diagnosable *in P iff*

$\mathcal{R} \neq \infty)$ $P^2_{loop} \times S_{obs\_eq} \models G(\neg loop \wedge \beta_L \rightarrow F(H^{\leq \mathcal{R}}(obs\_eq) \rightarrow O\beta_R))$

$\mathcal{R} = \infty)$ $P^2 \times S_{obs\_eq} \models G(\beta_L \rightarrow F(H(obs\_eq) \rightarrow O\beta_R))$

i.e., the coupled twin-plant constructed from P satisfies the Finite Delay property from Table 7.1 (for perfect recall) or Table 7.3 for bounded recall.

Proof. The proof follows the same pattern of bounded delay, with the exception that k and l are not bounded by the delay. □

7.2 Pareto Optimal Sensor Placement

If a plant is not system diagnosable for a specification, we can either weaken the specification (e.g., trace diagnosability) or try to increase the reasoning power of the diagnoser. In particular, we might want to increase its recall and the set of observables. The problem of finding a subset of sensors that makes the system diagnosable is called *sensor placement*. Substantial interest has been devoted to this problem, especially in the setting of autonomous systems, where we face trade-offs between the available resources (space, time, energy, etc.) and the accuracy of the diagnosis. Therefore, one of the design goals is to identify the subset of observables that makes the system diagnosable while minimizing some cost function. Notice that a cost can be also associated to the delay[2] that we provide to the diagnosis: we might be able to use less sensors [26] by allowing a longer delay. In many practical situations, there are multiple cost functions that we are trying to minimize at the same time, and often these are incomparable. In

[2]For consistency, we assume that we want to minimize all costs, however, the delay usually grows as an inverse of the number of observables.

these cases, the notion of *Pareto optimal* is commonly used. A solution is Pareto optimal w.r.t. a set of cost functions, if any solution that improves the value of one of the cost dimensions also worsens the value of another cost dimension. The set of solutions that are Pareto optimal is also called *Pareto frontier*.

In this section we formulate the problem of sensor placement with multiple cost functions. In particular, we will focus on the use of the delay as a cost function. We present three related approaches to solving this problem, and provide an experimental evaluation that also extends to other domains in which Pareto optimal parameter synthesis can be applied.

7.2.1 Problem Description

The problem of parameter synthesis (Section 2.5) aims at finding a valuation for a set of parameters, s.t. a given property is satisfied. In this chapter, we assume parameters to be Boolean, and call the set of valuations (i.e., possible values for the parameters) $\Gamma \stackrel{\Delta}{=} \mathbb{B}^{|U|}$. Given a parametric LTS $S = (V, E, U, I, T)$, each valuation of the parameters γ induces an LTS $S_\gamma = \langle V, E, \gamma(I), \gamma(T)\rangle$, in which we modify the initial condition and transition relation by replacing each parameter occurrence with its value. The order relation $<$ over $\mathbb{B}$ induces a partial order $\prec$ over the parameter valuations Γ.

ExactDel and BoundDel properties can be reduced to invariant properties. FiniteDel, instead, requires an LTL property. Since we are interested in seeing how the delay is influenced by the choice of sensors, in the rest of the Section, we assume that we are dealing with invariants. This assumption also helps from the practical point of view, since there is more research and tools that can solve the parameter synthesis problem for invariants than for full LTL.

Cost functions are integer-valued functions over parameter valuations:

$\textsc{Cost} : \Gamma \to \mathbb{N}$. A multi-dimensional cost function is defined as a vector of cost functions; for brevity we write $\textsc{Cost} : \Gamma \to \mathbb{N}^m$. We call $\mathbb{N}^m$ the space of costs. Given two cost vectors $(v_1, \cdots, v_m)$ and $(w_1, \cdots, w_m)$, we define the partial order relation $\preceq$ as $(v_1, \cdots, v_m) \preceq (w_1, \cdots, w_m)$ iff $\forall i.\ v_i \leq w_i$.

Given S, φ and $\textsc{Cost}$, we say that an assignment $\gamma \in \Gamma$ is Pareto-Optimal iff:

1. $S_\gamma \models \varphi$, i.e., it is solution and
2. there is no γ' s.t. $S_{\gamma'} \models \varphi$ and $\textsc{Cost}(\gamma') \prec \textsc{Cost}(\gamma)$.

The Pareto-Frontier $PF(\textsc{Cost}, \varphi) \subseteq \Gamma$ is the set of parameter assignments that are valid for φ and that are Pareto-optimal with respect to $\textsc{Cost}$. Pareto-optimality boils down to optimality with respect to a single cost function when $m = 1$. The cost function can be represented symbolically as a term $\textsc{Cost}(U)$; a set of assignments is then simply identified by a formula $\textsc{Cost}(U) \bowtie v$ where v is a natural number and $\bowtie$ is a relation operator. For example, $\textsc{Cost}(U) \leq 5$ are all the assignments with cost less than 5.

Monotonicity

We make two different assumptions of *monotonicity*: *property satisfaction* and *cost function*.

The first one is monotonicity of the satisfaction of the property. Intuitively, if the property holds under a given valuation, then it also holds for all the successors. Conversely, if the property does not hold for a given parameter valuation, then it does not hold for any of its predecessors. Formally, we say that $S \models \varphi$ is monotonic w.r.t. Γ iff

$$\forall \gamma,\ \text{If } S_\gamma \not\models \varphi \text{ then } \forall \gamma'.\ \gamma' \preceq \gamma \Rightarrow S_{\gamma'} \not\models \varphi$$

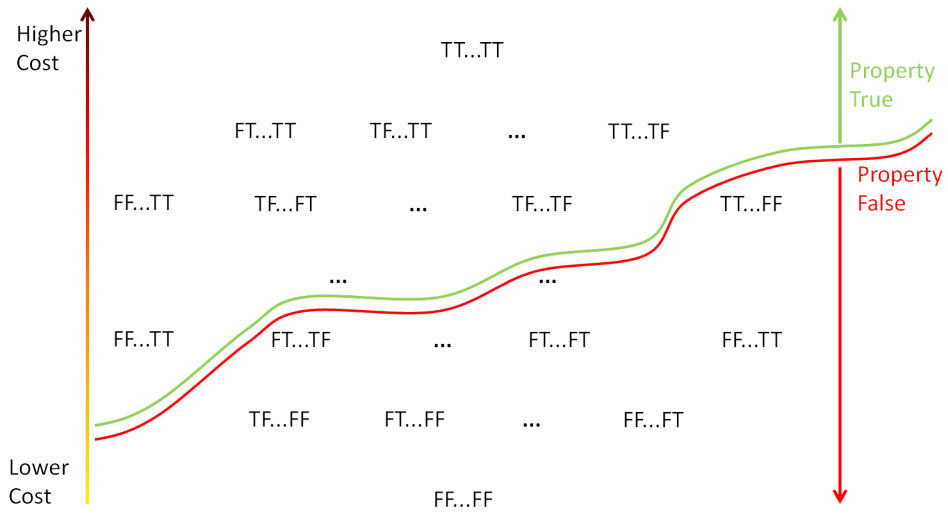

Figure 7.2: Monotonicity with respect to Property and Cost function

The second type of monotonicity is defined with respect to the cost function. Intuitively, costs follow the same ordering of the parameter valuation. Formally, we say that Cost is monotonic w.r.t. Γ iff

$$\forall \gamma, \gamma'. \text{ If } \gamma \preceq \gamma' \text{ then } \textsc{Cost}(\gamma) \preceq \textsc{Cost}(\gamma')$$

As depicted in Figure 7.2, the higher we go in the lattice of valuations, the higher the cost: the vertical arrow on the left denotes a cost function that is upwards monotonic along each path. The lines in the figure show the valuations for which the property φ holds (above the green) or not (below the red). These assumptions are important because they allow us to quickly prune the search space. For example, any time we find a valuation that does not satisfy the property, we can prune the search space by removing all valuations that are below it in the lattice. Similarly, we know that to reduce the cost we need to move downwards in the lattice.

We assume a two-dimensional cost function, although the approach should be extensible to cost functions of any dimension.

Approach Outline

We present several algorithms for the computation of the Pareto frontier, for a given S, φ, and Cost. These algorithms work under the assumption that both the property satisfaction relation ($S \models \varphi$) and Cost are monotonic with respect to Γ. To simplify exposition, we assume that there is at least one parameter valuation γ s.t. $S_\gamma \models \varphi$. This can be checked before starting the optimization problem.

The algorithms that we present define a spectrum based on how much of the set of ValidPars we compute up-front. In Section 7.2.2, we compute the whole set of good valuations up-front, and then proceed to the computation of the Pareto-Frontier. In Section 7.2.3, we *slice* the space ValidPars by one dimension and compute one of the slices at the time; once a slice has been computed, we minimize w.r.t. to the other costs. By doing so, we are able to skip slices (using the monotonicity assumption), so that we end up computing a subset of ValidPars. Finally, in Section 7.2.4 we do not compute ValidPars directly, but navigate through the valuations lattice driven by the cost functions and test on-the-fly membership of points to ValidPars.

7.2.2 The valuations-first approach

The first algorithm we present is an eager, two-stage approach. Figure 7.3 provides a high-level description of the algorithm.

function ValuationsFirst(S, Cost, φ)
  $VP :=$ ValidPars(S, φ)
  return ParetoFront(Cost, VP)
end function

Figure 7.3: Valuations First pseudo-code

The first stage constructs the set of parameter valuations that are valid

for the property φ. This gives a symbolic representation of the solution space (ValidPars), regardless of cost considerations. The second phase carries out an analysis of the solution space taking the costs into account, and selecting the assignments that are Pareto-optimal, thus building the Pareto front.

Each of the phases can be in turn refined. The computation of ValidPars can be carried out directly, by performing a reachability analysis on S, thus obtaining Reachable(U, V), and then considering only the valuations for which the states always satisfy the property. This idea has been explored with a BDD-based implementation in [41], where it was applied in the computation of Fault-Trees. In many cases, however, the computation of the reachable states can be over-killing.

function ValidPars(S, φ)
  $Bad := \bot$
  $S = (V, E, U, I, T)$
  while $S \not\models \varphi$ **do**
    $\gamma' :=$ project counter-example on U
    $Bad := Bad \vee \gamma'$
    $I := I \wedge \neg Bad$
  end while
  return $\neg Bad$
end function

Figure 7.4: ValidPars computation

In Figure 7.4 an alternative approach is presented, based on the algorithm proposed in [50], that constructs the set ValidPars $\stackrel{\Delta}{=} \{\gamma_i \mid S \models \gamma_i \rightarrow \varphi\}$ of valid parameter valuations. The idea is to rely on a model-checking routine to compute the set InvalidPars $\stackrel{\Delta}{=} \{\gamma_i \mid S \not\models \gamma_i \rightarrow \varphi\}$, i.e., a representation of the "lower part" of the lattice in Figure 7.2. At a very high level, this is done by enumerating counterexample traces to the negation of φ. Each trace is associated with an invalid parameter val-

uation, which is then accumulated in the result, and removed from the initial states, thus preventing the model checker from re-discovering it. Once $InvalidPars$ is computed, the space of valid parameter valuations is simply obtained by complement. This algorithm can thus rely on a model-checker as a black-box, therefore leveraging recent advancements in SAT-based model-checking techniques (e.g., IC3).

The second phase carries out the optimization of the combinational space defined by ValidPars with respect to Cost. This can be done, for example, by enumerating all the elements in ValidPars and comparing the associated costs, or by considering the symbolical characterization of the Pareto front:

$$\textsc{ParetoFront}(U) = \textsc{ValidPars}(U) \wedge \nexists U'.((U' \prec_{\textsc{Cost}} U) \wedge \textsc{ValidPars}(U'))$$

Computing ParetoFront(U) can be done via encoding of the cost functions into SAT or SMT, and application of optimization techniques [87, 143, 28].

The main drawback in the valuations-first algorithm is that in order to compute ValidPars, we need to enumerate all the elements of InvalidPars. This computation might be very expensive, and no information is available until this phase has been completed. In particular, we are not able to approximate the solution.

7.2.3 The One-Cost Slicing Approach

The second algorithm that we propose (Figure 7.5) interleaves the analysis of the cost information with checks on the validity of the parameters. This is done by slicing the space of valid parameters along the different dimensions (i.e., cost functions).

We first initialize c_1 to the highest possible value, and the Pareto frontier to the empty set. We iterate as follows. First, we compute all the candidate

function SLICING(S, COST, φ)
  PF := $\emptyset$
  $\gamma = \top$;
  $c_1 := \text{COST}_1(\gamma)$
  $S' := \text{FIXCOST}(S, \text{COST}_1 = c_1)$
  $VP_{\text{COST}_1} := \text{VALIDPARS}(S', \varphi)$
  while $VP_{\text{COST}_1} \neq \emptyset$ **do**
    $(\gamma, c_2) = \text{MINIMIZE}\ (\text{COST}_2, VP_{\text{COST}_1})$
    $(\gamma, c_1) := \text{REDUCE}_{\text{COST}_1}(S, \gamma, \varphi, c_2)$
    PF.add(γ, c_1, c_2)
    $c_1 := c_1 - 1$
    $S' := \text{FIXCOST}(S, \text{COST}_1 = c_1)$
    $VP_{\text{COST}_1} := \text{VALIDPARS}(S', \varphi)$
  end while
  return PF
end function

function FIXCOST(S, *CostExpr*))
  $S = (V, E, U, I, T)$
  $S' := (V, E, U, I \wedge CostExpr, T)$ **return** S'
end function

Figure 7.5: Slicing algorithm

solutions on the parametric system S' in which we fixed the cost COST_1 to the value c_1. We then search in the set of candidates (VP_{COST_1}) for the best valuation and cost for COST_2. Once an optimal cost c_2 has been found, we fix it and try to find a smaller valuation w.r.t. COST_1, and add the solution to the Pareto front. This is done by calling a function $\text{REDUCE}_{\text{COST}_1}$ which, given a solution γ of cost (c_1, c_2), returns another solution γ' of cost (c_1', c_2) with $c_1' \leq c_1$. For now, REDUCE is simply a function that tries to improve a candidate solution γ. Its actual implementation is described in the next Section.

In order to find the other points of the Pareto frontier, we decrease c_1

and test whether any solution (independently of COST_2) exists. If it does, we repeat the process, otherwise we terminate.

Note that in the function MINIMIZE we operate on the set of the solutions, while in REDUCE, we generate a candidate $\gamma' \preceq \gamma$ and test whether it is still a solution (i.e. $S_{\gamma'} \models \varphi$). Due to the monotonicity assumption, REDUCE cannot skip solutions. However, REDUCE can drastically accelerate the search by avoiding the enumeration of all costs in c_1.

In the pseudo-code, the addition of solutions to the Pareto front is simplified. We cannot add a solution γ_1 immediately in the Pareto front, but we need to wait for the next solution γ_2 to be added ($PF.add$). If the costs of γ_1 and γ_2 are incomparable, then γ_1 is Pareto-optimal and gets added to the frontier. If γ_2 is smaller than γ_1, then γ_1 is not optimal and is discarded. This pair-wise operation guarantees that only Pareto optimal solutions are considered.

This approach only computes slices of VALIDPARS when needed. Although in the worst-case we can end-up computing it all by slices, when calling the REDUCE function, it is generally possible to accelerate the search and skip intermediate slices.

7.2.4 The costs-first approach

One of the key insights of the slicing algorithm is that big parts of VALIDPARS might not be necessary in order to compute the Pareto front. In the costs-first approach we take this idea to the extreme: we do no compute VALIDPARS anymore. Instead, we explore the lattice of valuations induced by the cost functions. Every time we find a promising valuation γ, we test whether it is actually a solution (i.e., $S_\gamma \models \varphi$). Due to the monotonicity assumption, whenever we find a valuation that is not a solution, we can prune all of its predecessors in the lattice (since they cannot be solutions either).

```
function CostsFirst(S, Cost, φ)
    PF := ∅
    γ := ⊤;
    c1 = Cost1(γ); c2 = Cost2(γ)
    repeat
        c2 = c2
        for γi ∈ MaxSmallerCandidate_Cost2(c1, c2) do
            if S_γi ⊨ φ then
                (γ, c2) := Reduce Cost2(S, γ, φ, c1)
            end if
        end for
        (γ, c1) := Reduce Cost1(S, γ, φ, c2)
        PF.add(γ, c1, c2)
        c1 := c1 − 1
    until No solution exists for FixCost(S, Cost1 = c1)
    return PF
end function
```

Figure 7.6: CostsFirst pseudo-code

An overview of the algorithm is provided in Figure 7.6. We start by getting an upper-bound on both costs by considering the cost of the top valuation. In the outer-loop we decrease the value of $\textsc{Cost}_1$, similarly to the slicing approach. Within the inner-loop, however, we proceed by enumerating the solutions that have smaller value w.r.t. $\textsc{Cost}_2$. In particular, MaxSmallerCandidate returns the maximal solution(s) with the same cost c_1 but with smaller c_2. The process terminates whenever no solution can be found for a given value of $\textsc{Cost}_1$. Note how the structure of this algorithm is similar to the one of the slicing approach. The main difference is that we never need to compute ValidPars.

This algorithm has an additional advantage over the previous approaches: it can be interrupted at any point and is guaranteed to provide us with an under-approximation of the Pareto frontier. In fact, at any point in time, we have a subset of the Pareto frontier. We call this prop-

erty *any-time*. This is in contrast with the other approaches such as [41], or the valuations first approach, that require termination of the procedure in order to provide the solution space of the parameters.

7.2.5 IC3-based implementation

The approaches described in the previous section have been implemented on-top of an IC3-based engine. In particular, there are two key ideas that we can leverage in order to have an efficient implementation using IC3. First, we notice that $S_\gamma \models \varphi$ holds iff $S \models \gamma \rightarrow \varphi$. This observation makes it possible to reason always on the same system, and moves the choice of the valuations within the property. This leads us to the second fundamental observation. If $S \models \gamma \rightarrow \varphi$, we can extract from the IC3 model-checker the inductive invariant ψ. By definition of inductive invariant we know that $\psi \models \gamma \rightarrow \varphi$; moreover, it might be the case that we can reuse the same invariant to check whether another valuation γ' is a solution: i.e., $\psi \models \gamma' \rightarrow \varphi$. We will use this idea when trying to reduce the valuation, since this makes it possible to reason locally on a (relatively small) formula, and does not require unrolling or computing reachable states. The efficiency of the procedure will then largely depend on how well the reduction step works.

Figure 7.7 presents the adaptation of the costs-first algorithm when using the inductive invariant to perform the Reduce step. The same idea for Reduce can be applied in the slicing algorithm.

We navigate the lattice by picking the maximal candidate(s) of smaller cost w.r.t. Cost_2 (MaxSmallerCandidate). This fact guarantees that the algorithm will terminate, since we are always picking a solution of smaller dimension. We then check that the property still holds for the new valuation γ_i, by using IC3. If this is the case, we are provided with an inductive invariant ψ, s.t., $\psi \models \gamma_i \rightarrow \varphi$.

```
function CostsFirstIC3(S, Cost, φ)
    PF := ∅
    γ := ⊤;
    c_1 = Cost_1(γ); c_2 = Cost_2(γ)
    repeat
        c_2 := c_2
        for γ_i ∈ MaxSmallerCandidate_Cost_2(c_1, c_2) do
            (res, ψ) := IC3(S, γ_i → φ)
            if res == Safe then
                # ψ is an inductive invariant s.t. ψ ⊨ γ_i → φ
                (γ_i, c_1, c_2) := Reduce_Cost_2(ψ, γ_i, φ)
            end if
        end for
        (γ_i, c_1, c_2) := Reduce_Cost_1(ψ, γ_i, φ)
        PF.add(γ, c_1, c_2)
        c_1 := c_1 − 1
    until No solution exists for FixCost(S, Cost_1 = c_1)
    return PF
end function
```

Figure 7.7: IC3-based CostsFirst pseudo-code

The operation of picking a cost-predecessor could be, in principle, delegated to a pseudo-Boolean constraint solver, or to other reasoning engines that are able to deal with costs natively. For our simple implementation, we use an SMT solver with the theory of bit-vectors.

When considering the parameters as a set of elements, we can try to minimize the set by implementing the Reduce procedure using unsat-cores. Namely, we check the unsatisfiability of $\psi \wedge \neg\varphi$ under the assumption of γ_i and use standard features from modern SAT solvers to minimize the unsat-core that, in turn, translates in picking a subset of the parameters that makes the formula unsatisfiable. By doing so, we are able to "jump" and quickly reduce the valuation γ. For integer parameters, instead, we use a Reduce procedure that performs a linear or binary search, using the

inductive invariant. REDUCE could be the identity function, this will still preserve the correctness and termination of the approach. However, this would end-up requiring an explicit state search of the lattice. Having a smart REDUCE procedure makes it possible to jump and terminate faster. Other choices of REDUCE might be studied in the future.

Since the inductive invariant does not depend on the costs, it is possible to reuse the invariant from previous calls in an incremental way. Intuitively, this provides us with stronger invariants that are more likely to allow us to reduce the parameters aggressively. In particular, at iteration i, we obtain the invariant ψ_i, that can be strengthen by considering $\psi'_i = \bigwedge_{n\in[0,i]} \psi_n$.

7.2.6 Experimental Evaluation

The algorithms described in this section were implemented on top of the nuXMV model checker [47].[3] We evaluate our approach on a series of benchmarks from the domains of sensor placement [26]. These examples were obtained from realistic case studies on a simpler problem, i.e. finding a good set of sensors for a fixed delay. These simpler benchmarks were challenging and in some case not solvable by previous techniques [26].

Some benchmarks come from the aerospace domain: ORBITER, ROVERSMALL, and ROVERBIG (from [40]), CASSINI [41], X34 [55]. ELEVATOR models an elevator controller, parameterized by the number of floors. C432 is a Boolean circuit used as a benchmark in the DX Competition [82]. All models also contain faults based on which a sensor placement problem is formulated.

Product Lines When designing a product, engineers are often faced with a high degree of variability in terms of possible features. Such variability is

[3]The executable and benchmark instances used in the evaluation are available at `http://marco.gario.org/phd/`

usually captured in *product line* models (sometimes referred to as feature models). For instance, in [92] the authors model variability in a controller design, and the authors of [60] consider software product lines. Here we are specifically interested in the analysis of dynamic systems as opposed to static contexts which are usually addressed with constraint programming techniques.

The goal of product line engineering is usually to identify which combinations of features satisfy a certain property. Here however we specifically address the Pareto-optimal trade-off problem. In various works there are different assumptions on the monotonicity of features, that is whether by adding features the possible behaviors increase monotonically or whether some behaviors can be overridden. In our work we only assume the monotonicity of the property of interest in terms of feature additions.

The properties for the product lines benchmarks, that were derived for our invariant checking framework, are artificial but tailored specifically for these examples. For both cases we are unfortunately not aware of other publicly available industrial benchmarks. PRODUCT LINES are benchmark instances derived from [92], describing a railway switch controller. The product-line features correspond to possible communication strategies used by the controller. We explore a design trade-off along two dimensions. The first dimension is the upper bound on the message sequence length. The second dimension is the cost associated to dropping a particular feature, specified by a random cost function. Our aim is to minimize both the message sequence bound and the cost of removed features.

Results For each example, we generated multiple benchmarks by varying both the number of parameters considered and the (randomly-generated) costs of the individual parameters. Overall, our benchmark set consists of 81 instances. The number of Pareto-optimal solutions varies between 0 and

5. Experiments were executed on a Linux cluster equipped with 2.5Ghz Intel Xeon CPUs with 96Gb of RAM. We used a time limit of 1 hour and a memory limit of 6Gb.

Family	#Instances	valuations-first	one-cost slicing	costs-first
c432	32	11	13	**32**
cassini	21	6	12	**21**
elevator	4	4	4	4
orbiter	4	4	4	4
roversmall	4	4	4	4
roverbig	4	4	4	4
x34	4	4	4	4
product lines	8	6	4	**8**
TOTAL	81	43	49	**81**

Figure 7.8: Number of solved instances by each approach

In Figure 7.8 we present the number of instances solved for each problem family. For the c432, cassini and product lines benchmarks, we can see how the costs-first approach finds all the solutions within the timeout, whereas the other two approaches fail on several instances. Figure 7.9 shows the accumulated-time plots for the different algorithms, plotting the number of solved instances (y-axis) in the given total amount of time (x-axis) in logarithmic scale.

For the c432 and cassini benchmark, we show in Figure 7.10 the runtime as a function of the parameters. As expected, on the same model, the number of parameters has a big impact on the runtime. Indeed, for the valuations-first and the one-cost slicing approaches this has an exponential tendency.

Finally, in order to evaluate the impact of the Reduce procedure in the costs-first algorithm, we performed an experiment in which we ran costs-first without applying Reduce. From the scatter plot of Figure 7.11, we can see that Reduce is crucial for performance: without it, costs-first

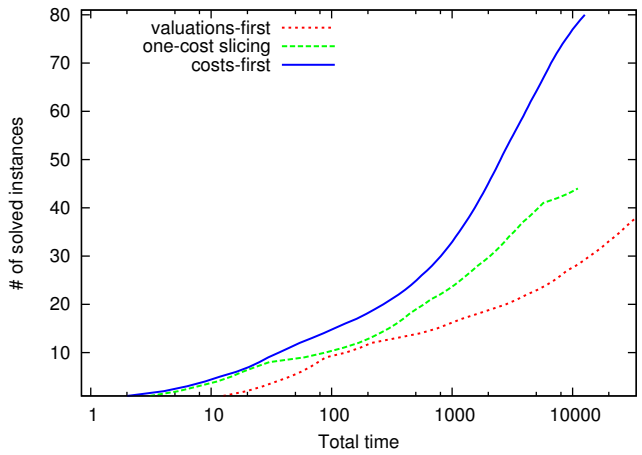

Figure 7.9: Accumulated-time plot showing the number of solved instances (x-axis) in a given total time (y-axis) for the various algorithms.

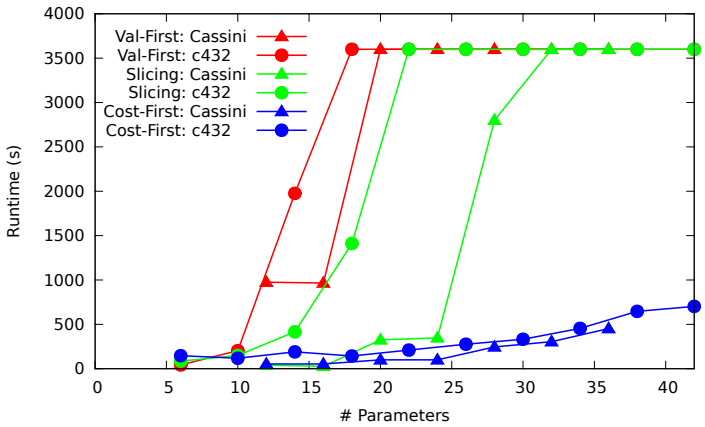

Figure 7.10: Runtime for different number of parameters

solves only 48 (out of 81) instances, and the runtimes increase by orders of magnitude.

7.3 Chapter Summary

In this chapter we discussed the twin-plant approach for (system) diagnosability testing. We show how to extend the twin-plant approach to deal with different types of alarm conditions and different amounts of recall. By using the twin-plant approach, we can leverage parameter synthesis

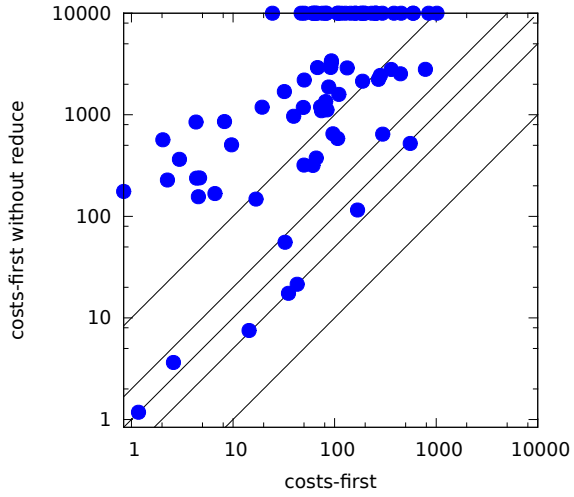

Figure 7.11: Impact of REDUCE in the costs-first algorithm.

engines in order to perform synthesis of observables. We discussed several approaches for performing Pareto-optimal parameter synthesis under the assumption of monotonicity. In particular, we showed that our IC3-based implementation makes it possible to solve many instances if we enable symbolic pruning techniques.

Chapter 8

Synthesis

Manually designing an FDI is a difficult task. Minor changes in the plant design might brake the current FDI design. For this reason, the design of the FDI(R) is commonly postponed until late in the system development life-cycle. The idea is to wait until a point in which the plant design is stable, before starting the FDI design. This process, however, results in a drastic shortening of the time available to design, implement and test the FDI. The outcome is that the FDI might be shipped when still partially incomplete.

In the previous chapters we discussed how to carry out the specification and validation of the FDI requirements. By having formal requirements it is easier to drive the verification of the FDI, and continuously verify that the system provides enough information for the requirements to be implemented (diagnosability).

In this chapter, we take a step forward, and discuss the possibility of automatically synthesizing an FDI starting from the formal requirements (ASL_K specification) and a formal model of the system. The advantages of FDI synthesis are many-fold. First, the time required to get a working FDI is reduced considerably. Second, the design of the FDI does not require an expert knowledge of the system. Third, additional optimization constraints

on the design can be imposed, e.g., limit the size of the FDI or the access to sensors. Finally, the synthesized FDI can be used as reference while manually designing a new one.

In this chapter, we discuss the synthesis process for finite state systems under perfect recall (Section 8.1), and for both finite and infinite state systems under bounded recall (Section 8.2).

The contribution of this Chapter are:

- An algorithm for the synthesis of perfect-recall diagnosers for finite state systems (originally presented in [34, 33])
- An algorithm for the synthesis of bounded-recall diagnosers based on parameter synthesis
- Examples of the application of both, and experimental data on their applicability.

8.1 Perfect Recall

In this section, we discuss how to synthesize a perfect recall diagnoser that satisfies a given specification $\mathcal{A}$. The choice of perfect recall as a starting point is driven by the existing literature [138, 140], and by the fact that perfect recall diagnosers are more powerful than bounded recall ones. We consider the most expressive case of ASL_K (maximal/trace diagnosable), which also satisfies all the other cases, and we limit the discussion to finite state systems. A the end of the section, we will discuss the challenges involved in extending the approach to infinite state systems. We formally prove several formal properties of the synthesized diagnoser, and in particular that it satisfies the specification.

The idea of the synthesis algorithm is to generate an automaton that encodes the set of possible states in which the plant could be after each

observation. The result is achieved by generating the power-set of the states of the plant, also called *belief states*[1], and defining a suitable transition relation among the elements of this set, only taking into account observable information. Each belief state of the automaton is then annotated with the alarms that are satisfied in all the states of the belief state. The resulting automaton is the Diagnoser.

8.1.1 Synthesis algorithm

Given a partially observable plant $P = \langle V^P, E^P, I^P, T^P, E^P_O \rangle$, let S be the set of states of P. The *belief automaton* is defined as $\mathcal{B}(P) = \langle B, E, b_0, R \rangle$ where $B = 2^S$, $E = E^P_o$, $b_0 \in B$ and $R : (B \times E) \to B$. B represents the set of sets of states, also called *belief states*. Given a belief state b, we use b^* to represent the set of states that are reachable from b by only using events in $E^P \setminus E^P_o$ (non observable events), and call it the u-transitive closure. Formally, b^* is the least set s.t. $b \subseteq b^*$ and if there exist $e \in E^P \setminus E^P_o$ and $s' \in b^*$ such that $\langle s', s \rangle \in \mathcal{T}^P(e)$ then $s \in b^*$. b_0 is the initial belief state and contains the states that satisfy the initial condition I^P (i.e., $b_0 = \{s \mid s \models I^P\}$).

Given a belief state b and an observable event $e \in E^P_o$, we define the successor belief state b' as:

$$R(b, e) = b' = \{s' \mid \exists s \in b^*.\ \langle s, s' \rangle \models T^P(e)\}$$

that is the set of states that are compatible with the observable event e in a state of the u-transitive closure of b. Intuitively, we first compute the u-transitive closure of b to account for all non-observable transitions, and then we consider all the different states that can be reached from b^* with an occurrence of the event e.

[1]The name *belief state* comes from the setting of planning under partial observability. In our epistemic setting, it would be more accurate to name them *knowledge state*, since we are dealing with knowledge and not belief. However, for consistency with previous work we maintain the name *belief state*.

The diagnoser is obtained by annotating each state of the belief automaton with the corresponding alarms. We annotate with A_φ all the states b that satisfy the temporal property $\tau(\varphi)$. As explained later on, any temporal $\tau(\varphi)$ can be handled by introducing suitable propositional formulas. Therefore we consider the simplest case in which $\tau(\varphi)$ is a propositional formula and formally say that the annotation a_b of the belief state b is the assignment to A_φ such that $a_b(A_\varphi)$ is true iff for all $s \in b$, $s \models \tau(\varphi)$. We perform the same annotation for $A_{\neg\varphi}$. The diagnoser obtained by this algorithm induces three alarms, related to the knowledge of the diagnoser. In particular, the diagnoser can be sure that a condition occurred (A_φ) can be sure that a condition did not occur ($A_{\neg\varphi}$) or can be uncertain on whether the condition occurred ($\neg A_\varphi \wedge \neg A_{\neg\varphi}$) – notice that, by construction, it is not possible for both A_φ and $A_{\neg\varphi}$ to be true at the same time. In this way, at any point in time we are able to understand whether we are on a trace that is not diagnosable (and thus there is uncertainty) or whether the diagnoser knows that the condition did not occur. This can provide additional insight on the behavior of the system.

Figure 8.1 provides a pseudo-code of the main function of the synthesis task: the construction of the belief automaton. Starting from the set of initial states, we perform an explicit visit until we have explored all belief states. For each belief state we first compute its u-transitive closure ($u\_trans\_closure$) w.r.t. the non-observable events E, obtaining b^*. We then compute the possible observable events available from b^*, and iterate over each event o_i obtaining the set of states $target\_belief$ such that $T(b^*, o_i, target\_belief)$ is satisfied ($reachable\_w\_obs$). We can now add a transition to our automaton linking the belief state b to the belief state $target\_belief$ through the event o_i. Once we have completed this phase, we have an automaton with labeled transitions. The automaton resulting from this function can then be annotated by visiting each state and testing

```
function BELIEF_AUTOMATON(I, T, E, E_o)
    visited ← {}
    edges ← {}
    stack ← [I]
    while not stack.is_empty() do
        b ← stack.pop()
        b* ← u_trans_closure(b, T, E)
        for all o ∈ get_observable_events(b*, T, Eo) do
            target_belief ← reachable_w_obs(b*, o, T)
            edges.add((b, o, target_belief))
            if target_belief ∉ visited then
                visited.add(target_belief)
                stack.push(target_belief)
            end if
        end for
    end while
    return Automaton(visited, edges)
end function
```

Figure 8.1: Pseudo-code of the Belief Automaton construction phase

whether the state entails (or not) the alarm specification.

The approach resembles the constructions by Sampath [138] and Schumann [140], with the following main differences. First, we consider LTL past expression as diagnosis condition, and not only fault events as done in previous works. Second, instead of providing a set of possible diagnoses, we provide alarms. In order to raise the alarm, we need to be certain that the alarm condition is satisfied for all possible diagnoses. This gives raise to a 3-valued alarm system: we *know* that the fault occurred; *know* that the fault did *not* occur; or we are *uncertain*.

The approach also resembles the subset construction used in [69, 153] for model-checking temporal epistemic logic. This fact is yet another indication that there is a strong relation between temporal epistemic logic and FDI design.

8.1.2 Formal Properties of the Synthesized diagnoser

For each alarm condition, the temporal condition is defined as (see Section 6.1):

- $\tau(\varphi) = Y^d\beta$ for $\varphi = \textsc{ExactDel}(A, \beta, d)$;
- $\tau(\varphi) = O^{\leq d}\beta$ for $\varphi = \textsc{BoundDel}(A, \beta, d)$;
- $\tau(\varphi) = O\beta$ for $\varphi = \textsc{FiniteDel}(A, \beta)$.

notice that those are all safety property. The results below hold for any of those τ.

The generated transition system is a correct, complete and maximal diagnoser. We build a new plant P' by adding a monitor variable $\overline{\tau}$ to P s.t., $P' = P \times (G(\tau(\varphi) \leftrightarrow \overline{\tau}))$, where we abuse notation to indicate the synchronous composition of the plant with an automaton that encodes the monitor variable. In this way, we can handle the three alarm conditions in

the same way, by reducing them to a zero-delay alarm. In fact, the alarm condition is rewritten as $\varphi' = \textsc{ExactDel}(A_\varphi, \overline{\tau}, 0)$.

$$D \otimes P \models \varphi \text{ iff } D \otimes P' \models \varphi'$$

We define D_φ as the diagnoser for φ. $D_\varphi = \langle V^{D_\varphi}, E^{D_\varphi}, I^{D_\varphi}, T^{D_\varphi} \rangle$ is a symbolic representation of $\mathcal{B}(P)$ with $A_\varphi \subseteq V^{D_\varphi}$, $E_o^{D_\varphi} = E_o^P$ and such that every state b of D_φ represents a state in B (with abuse of notation we do not distinguish between the two since the assignment to A_φ is determined by b).

Theorem 12. *D_φ is deterministic.*

Proof. The result follows directly from the definition of the belief automaton, which is deterministic (one initial state and one successor). The assignment to A_φ is determined by the belief state. □

Lemma 1. *For every reachable state $b \times s$ of $D_\varphi \otimes P$, for every trace σ reaching $b \times s$, for every state $s' \in b$, there exists a trace σ' reaching $b \times s'$ with $obs(\sigma) = obs(\sigma')$.*

Proof. By induction on σ. All traces are observationally equivalent in the initial state. Let $\langle b_1 \times s_1, e, b \times s \rangle$ be the last transition of σ and let σ_1 be the prefix of σ without this last transition. If $e \in E \setminus E_o$ then $obs(\sigma) = obs(\sigma_1)$. Otherwise, for every state $s' \in b$ there exists a transition $\langle s'_1, e, s' \rangle$ such that $s'_1 \in b_1^*$. By inductive hypothesis there exists a trace σ'_1 reaching $b_1 \times s'_1$ such that $obs(\sigma_1) = obs(\sigma'_1)$. Therefore the concatenation of σ'_1 with the transition $\langle b_1 \times s'_1, e, b \times s' \rangle$ results in a trace σ' reaching $b \times s'$ such that $obs(\sigma) = obs(\sigma')$. □

Theorem 13 (Maximality)**.** $D_\varphi \otimes P \models G(\llcorner K\tau(\varphi) \lrcorner \rightarrow \llcorner A_\varphi \lrcorner)$.

Proof. Consider a trace σ and $i \geq 0$. If $\sigma, i \models \llcorner K\tau(\varphi) \lrcorner$, then for all traces σ' and points j s.t. $ObsEq((\sigma, i), (\sigma', j))$, $\sigma', j \models \tau(\varphi)$ (Recall that $\llcorner \cdot \lrcorner$ indicates

an observation point – Definition 10). By Lemma 1, all states $s \in \sigma[i]$ there exists a trace σ' with $obs(\sigma) = obs(\sigma')$, and therefore $s \models \tau(\varphi)$ so that $\sigma[i] \models \llcorner A_{\varphi} \lrcorner$. □

Lemma 2. *Given a trace σ of $D_\varphi \otimes P$. Let $\sigma[i] = b \times s$. If i is an observation point, then $s \in b$.*

Proof. By assumption, i is the n-th observation point of σ for some n. We prove the lemma by induction on n.

Consider the case $n = 1$. If $\sigma[0] = b_0 \times s_0$, by construction of D_φ, $s_0 \in b_0$. Let $\sigma[i-1] = b' \times s'$ and let e be the i-th (observable) event of σ. If i is the first observation point of σ, it means that $b' = b_0$ and $s' \in b_0^*$. Moreover, $\langle s', s \rangle \in T(e)$ and therefore $s \in b$.

Consider the case $n > 1$. Let j be the $n-1$ observation point, $\sigma[j] = b_j \times s_j$, $\sigma[i-1] = b' \times s'$ and let e be the i-th (observable) event of σ. Similarly to the previous case, $b' = b_j$ and $s' \in b_j^*$. Moreover $\langle s', s \rangle \in T(e)$ and therefore $s \in b$. □

Theorem 14 (Correctness)**.** $D_\varphi \otimes P \models G(\llcorner A_{\varphi} \lrcorner \rightarrow \tau(\varphi))$.

Proof. Consider a trace σ and $i \geq 0$. Suppose $\sigma, i \models \llcorner A_{\varphi} \lrcorner$ and let σ_{D_φ} and σ_P be respectively the left and right component of σ. Then, for all $s \in \sigma_{D_\varphi}[i]$, $s \models \tau(\varphi)$. Since i is an observation point, by Lemma 2, $\sigma_P[i] \in \sigma_{A_\varphi}[i]$. We can conclude that $\sigma[i] \models \tau(\varphi)$. □

Theorem 15 (Completeness)**.** *If φ is an alarm condition required to be trace diagnosable, then D_φ is complete. If φ is a system diagnosable condition and φ is diagnosable in P, then D_φ is complete.*

Proof. Since D_φ is maximal and correct (Theorems 13 and 14), we can apply Theorem 6 (if φ is trace diagnosable) or Theorem 7 (if it is system diagnosable) to obtain completeness. □

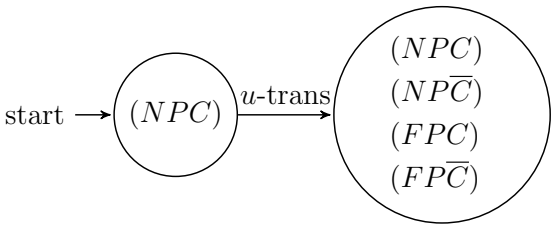

Figure 8.2: Expanding the initial belief state of the battery LTS.

8.1.3 Battery Sensor System

We show the first step of the algorithm on a simplified version of the battery component of our running example (Figure 4.9). We ignore the events related to threshold passing of the battery (*Mid*, *Low*, *High*) and only consider the observable event *Off*, signaled when the charge reaches zero, and the ones due to mode changes. To keep the representation compact, we indicate each state with three symbols. For example, we use (NPC) to indicate the state "Nominal, Primary, Charging" and $(NP\overline{C})$ to indicate the state "Nominal, Primary, Not Charging". Similarly we use F, O, and D to indicate Faulty, Offline and Double. Recall that in the original model, the mode transitions are observable but all other transitions are not.

In the first step (Figure 8.2), we take the set of initial states. This is the set of states (NPC) for any value of the $charge \in [0, C]$. The u-transitive closure needs to take into account all non-observable transitions. Therefore, we need to consider going from Nominal to Faulty, from Charging to Not Charging, and their combination.

These are all the states that are reachable before an observable event can occur. We now take each observable event and compute the set of states that are reachable with one of the observable events (Figure 8.3): the battery being discharge (Off), and the change of mode (*Offline*, *Double*). Note that one of the belief states (marked with ‡) is smaller than the others. This is due to the fact that in our model, the discharging of the

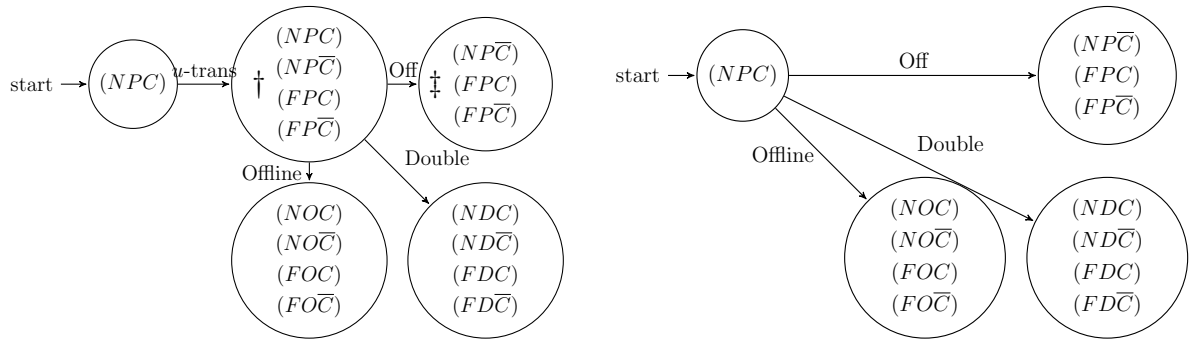

Figure 8.3: Expanding the belief state via observable transitions

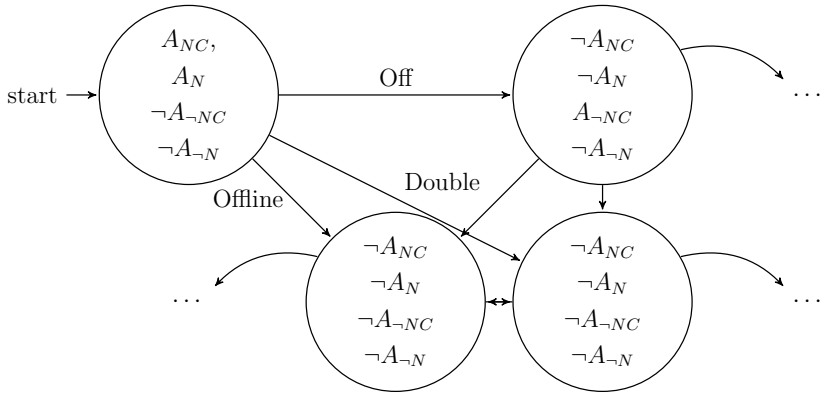

Figure 8.4: Annotation of the belief states

battery cannot occur if the battery is nominal, charging and in primary mode (NPC). Thus, the fact that we receive the Off event allows us to exclude that state. The state obtained by computing the transitive closure (marked with †) is not part of our final automaton, and is provided in the figure only to simplify the understanding.

We repeat these two steps until all belief states have been explored. We then proceed to the labeling phase, in which we label each state with the corresponding alarm. For example, by considering the alarms $\textsc{ExactDel}(A_{NC}, Nominal \wedge Charging, 0)$ and $\textsc{ExactDel}(A_N, Nominal, 0)$, we obtain the diagnoser partially represented in Figure 8.4. Notice how, in the initial state we can raise the alarm A_{NC}, and this alarm can only be changed by an observable transition.

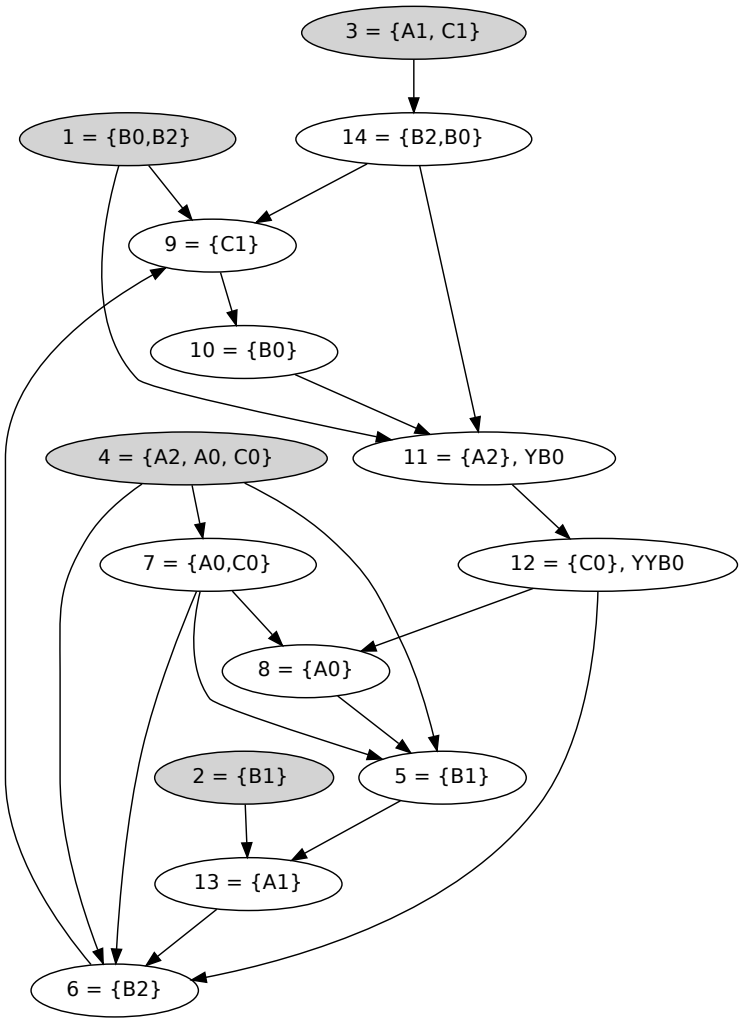

Figure 8.5: The belief automaton of the Magicbox example.

X	Y	State
0	0	$4 = \{\texttt{A2}, \texttt{A0}, \texttt{C0}\}$
0	1	$3 = \{\texttt{A1}, \texttt{C1}\}$
1	0	$1 = \{\texttt{B0}, \texttt{B2}\}$
1	1	$2 = \{\texttt{B1}\}$

Figure 8.6: Initial states observations

8.1.4 MagicBox

Figure 8.5 shows the belief automaton for our magicbox example (Figure 6.4), and the following specification $\mathcal{A}$:

$$\begin{aligned}\mathcal{A} = \{ & \\ & \varphi_1 = \textsc{ExactDel}_K(A(B1), \beta_{B1}, 0, system, True), \\ & \varphi_2 = \textsc{ExactDel}_K(A(C1), \beta_{C1}, 0, trace, True), \\ & \varphi_3 = \textsc{BoundDel}_K(A(B0), \beta_{B0}, 2, system, True) \\ \} & \end{aligned}$$

This example is smaller than the battery sensor. Therefore, we are able to show the complete construction. We work in the synchronous setting. To keep the diagram simple, we did not add the observables on the edges. Note that in the diagram states `2` and `5` have the same belief state but are handled differently because `2` is an initial state.

At the beginning we do not know where the ball is, therefore our first observation splits the initial belief state in 4 possible belief states (gray nodes) (Figure 8.6).

Let us take the most ambiguous initial state: `4`. This state has transitions to `5`, `6` and `7`. Each of these transitions will reduce the uncertainty on the location of the ball; in particular, in `5` and `6` we will have certainty on the ball location (`B1` and `B2` respectively). In `7` we can say for sure that the ball is in the column 0: $\texttt{7} \models (\texttt{A0} \vee \texttt{B0} \vee \texttt{C0})$.

Finally, note that `11` and `12` contain the monitor variables $YB0$ and $YYB0$, that will be used for handling the specification $\varphi_3 = \textsc{BoundDel}_K(A(B0), \beta_{B0}, 2, system, True)$.

Once we have constructed the belief automaton, we can navigate it and add the alarm A_φ whenever the temporal formula $\tau(\varphi)$ holds. The result of the annotation process is presented in Figure 8.7, where for clarity we show only the positive annotations. Let us take the first of our specifications: $\varphi_1 = \textsc{ExactDel}_K(A(B1), \beta_{B1}, 0, system, True)$. We annotate with $A(B1)$ all the states b in which $b \models \tau(\varphi_1)$, i.e., $b \models$ `B1`. These are the states `2` and `5`; all other states are marked with the negation of the alarm $!A(B1)$.

We proceed in a similar way for the second specification

$$\varphi_2 = \textsc{ExactDel}_K(A(C1), \beta_{C1}, 0, trace, True)$$

However, we realize that there is only one state in which `C1` holds, i.e., `9`. In `3` we have an ambiguity between `A1` and `C1`. If our specification required system diagnosability, `3` would be a witness of the non-diagnosability of the system. However, our specification requires trace diagnosability. Therefore, we annotate `9` with $A(C1)$ and all other states with $!A(C1)$. It should be clear, that using the same annotation ($!A(C1)$) for both `3` and (e.g.) `14` is counter-intuitive. In `3`, we do not know whether the ball is in `C1`, while in `14` we are *sure* that the ball is not in `C1`. To handle this situation, we break the specification φ_2 in two parts:

$$\varphi_{2+} = \textsc{ExactDel}_K(K(C1), \beta_{C1}, 0, trace, True)$$
$$\varphi_{2-} = \textsc{ExactDel}_K(K!(C1), \neg\beta_{C1}, 0, trace, True)$$

For clarity, we include the epistemic operator (K) in the alarm name, therefore encoding all situations of interest: the diagnoser *Knows* that `C1` holds, or it *Knows* that it does not. We can now annotate `9` with $K(C1)$, all

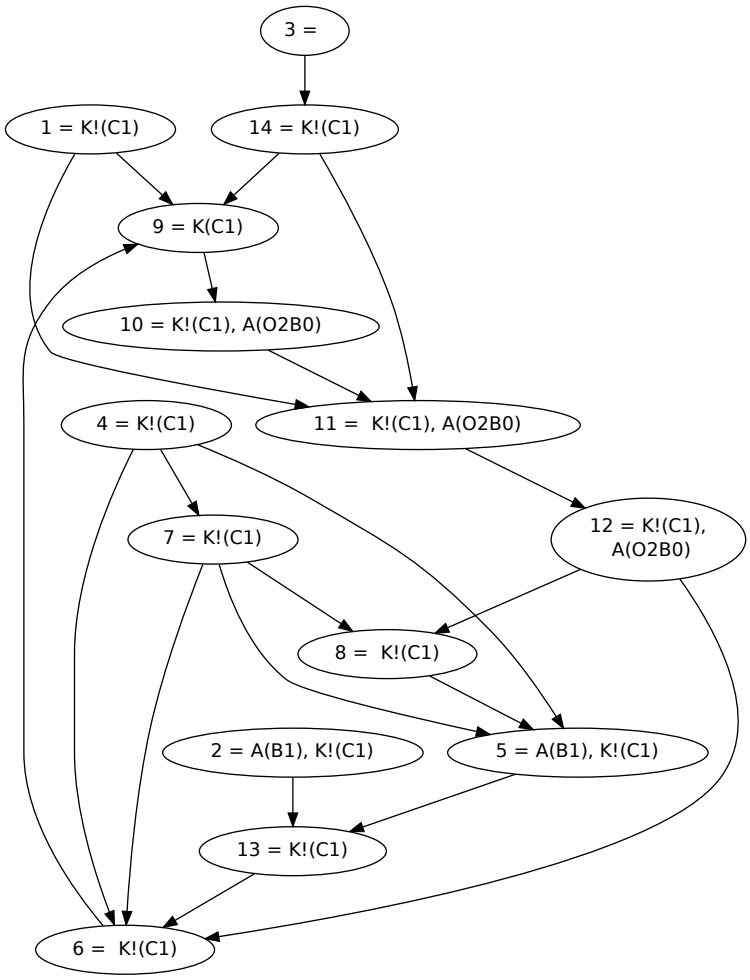

Figure 8.7: The diagnoser

the other states (except 3) with $K!(C1)$ and then complete the annotations with $!K(C1)$ and $!K!(C1)$. After performing the annotation, in 3 neither the alarm $K(C1)$ nor $K!(C1)$ will hold, due the non-diagnosability. This is particularly useful, because now we can react accordingly; for example, a recovery module might act in a different way knowing whether there is uncertainty on the occurrence of $C1$.

Splitting the specification in a positive and negative part is even more interesting for bounded specification. Let us consider $\varphi_3 = \textsc{BoundDel}_K(A(B0), \beta_{B0}, 2, system, True)$. The diagnosis condition is di-

agnosable with delay 2. We have uncertainty in 14, but then in 11 we know that $KO^{\leq 2}B0$ holds (due to the monitoring variable $YB0$). Therefore, the specification is system diagnosability. However, it might be interesting to have both positive and negative alarms, in order to annotate 14 and 1 with the information that $O^{\leq 2}B0$ *might* hold there, and distinguish them from all other states in which we are sure that it does not hold (e.g., 5).

Trace diagnosability is needed in the initial states. For example, we have uncertainty for $C1$ in 3, however, this is the only situation in which we have uncertainty on $C1$. Excluding this state, the rest of the system is diagnosable. This motivates the idea of trace diagnosability.

Figure 8.7 shows a maximal annotation for φ_3. The annotation $A(O2B0)$ is given to 10, 11 and 12. This means that, for each state in which $\tau(\varphi) = O^{\leq 2}\beta_{B0}$ holds, we have the annotation $A(O2B0)$. We could argue that having the alarm $A(O2B0)$ only in 11 would be correct too. Even more, we would still be correct if we had the alarm on 10 and 12 only. The three formalizations satisfy our definition of non-maximal bounded-delay specification; however, this situation leaves many details on the behavior of the diagnoser under-specified, making it hard to use the information provided by the diagnoser to devise a recovery or containment procedure. This is an example of how maximality provides a clear and un-ambiguous specification of the diagnoser.

8.1.5 Perfect Recall and Infinite State

In order to apply the belief explorer construction to an infinite state system, we need to overcome a few issues.

First, computing the forward image (i.e., to obtain successor states) requires performing a quantifier elimination. Even in the cases in which this is decidable, it is usually computationally expensive.

Second, an infinite state system might have an infinite number of belief

states. The simplest example is the case in which a variable keeps growing as a function of time. Thus, the procedure will not terminate. In order to mitigate this issue, we expect that the belief construction needs to integrate better the observables and the alarm conditions. Currently, the belief construction and state annotation phases are completely separated. By knowing which alarms and observables are needed to perform the annotations, the belief construction might be able to prune and simplify the belief states without loosing important information. In this way, it should be possible to collapse belief states that have reached a fix-point for what concerns the observations and alarms.

Finally, let us assume that we are able to generate the belief states for the system. In order to understand whether we have already visited a belief state (and thus reached a fix-point), we need a way to compare belief states. In the finite case, the use of BDDs provides a canonical representation of the belief states, thus we can detect that we have already visited a belief state in constant time. For infinite state systems, canonical representations are limited and thus to detect completion we would (in the worst case) perform $O(n)$ checks for n states of the belief automaton. Consider that in the finite state, the number of belief states is exponential in the number of states of the system (that, in turn is exponential in the number of variables); this means that adequate data-structures need to be identified to make the algorithm work in practice.

In the literature there has been some work that takes advantage of particular features of the underlying infinite state system, in order to perform the diagnoser construction. For timed automata, [151] provides an algorithm to construct the explicit diagnoser using Difference Bound Matrices [18] (DBM). In a timed automata, the only infinite component is represented by the clock variables. DBMs are commonly used for timed automata analysis. Therefore, the intuition of the approach is to decouple

the finite variables from the clock variables, and handle the latter with ad-hoc data structures.

Similarly, [149] proposes an algorithm for Stochastic Discrete Event Systems (SDES). A SDES has an infinite characterization due to the probability of reaching a state. The idea used in [149] is to decouple the state of the system from the probability function. Each edge of the resulting diagnoser is annotated with a probability matrix, that is used at runtime to update the probability of being in a given state within the target belief state.

Both [151] and [149] are mostly theoretical works, and it is unclear whether they can scale to models of realistic size. Nevertheless, both works suggest to divide the finite from the infinite part, and handle them separately. this approach might be applicable to other types of system.

Approaches based on the power-set construction are not the only option to perform synthesis. A different strategy, could involve exploring techniques from automata learning such as L^* [5] and variants [1] that are designed to learn Input/Output systems.

8.2 Bounded Recall

In some practical applications, the assumption of perfect recall is too strong. In practice, we expect the diagnoser to recall a limited number of observations and act upon those. This justifies the use of bounded recall for the diagnoser.

The outcome of the synthesis process in the perfect recall case is an automaton. In the bounded recall case, instead, it is a combinational circuit. This has several advantages, for example the fact that we can synthesize a diagnoser also for infinite state systems, and that we can verify properties of the diagnoser using SAT/SMT technologies.

8.2.1 Diagnoser Synthesis as Parameter Synthesis

In a bounded recall diagnoser with recall $\mathcal{R} \neq \infty$, an alarm A can be seen as an expression $A \subseteq O^{\mathcal{R}}$ stating which tuples of observations need to trigger the alarm. In particular, we want to find all the observations that satisfy the alarm specification. The alarm A, becomes a function of the observations in the last $\mathcal{R}$ time-steps. To simplify notation, we count backwards and indicate with o_0 the current observation, o_1 the observation 1 step before the current and so on. Therefore, the alarm A with $\mathcal{R} = 3$, becomes a relation $A \subseteq O^3$ and we can write it as $A(o_0, o_1, o_2)$.

Identifying the set of observations that belong to the alarm relation can be done through a parameter synthesis problem, in which we consider each observable as a parameter, and try to identify which values of the observations imply the occurrence of the diagnosis condition. The result of the synthesis process is a region (i.e., a set of observations) that satisfies the alarm condition. This requires introducing a parameter u_o for each of the observable variable o in the plant (u_o and o must have the same type). According to Definition 5, given the parametric transition system $P = (V, U, I_U, T_U)$, the parameter synthesis problem consists in identifying the region ω s.t. $\omega = \{\gamma \mid P_\gamma \models \gamma(\phi)\}$. The property ϕ in this case consist of the satisfaction of the temporal condition τ (Section 8.1.2):

$$\phi := G((\bigwedge_{o \in O} u_o = o) \rightarrow \tau(x))$$

To synthesize this region we proceed in two steps. First, given a finite sequence of observations, we compute the set of reachable states that are compatible with the observation. Second, we test whether the given set of reachable states satisfies a given alarm condition.

The first procedure is obtained by extending the plant with a queue of observations. We attach to the plant a FIFO (First In First Out) queue, in order to store the observations, and each time step we discard the ones that

are older than $\mathcal{R}$. In order to properly capture the semantics of bounded recall, we also need to add one bit of information telling us how many of the observations in the queue are real observations. This makes sure that we handle correctly the initial states of the queue. By using this queue, we are moving the problem from traces to states.

The second point has already been discussed in the perfect recall case, when labeling the belief states with the alarms that are satisfied or not. Given a set of states, we check whether they all satisfy the monitor for the temporal condition $\tau(\varphi)$ for the alarm condition φ.

Let $Queue_{\mathcal{R},E_O}$ be a transition system that stores the latest $\mathcal{R}$ observables events E_O. To synthesize the diagnoser, we proceed as follows:

1. Compute the set $Reach$ of reachable states of $P \otimes Queue_{\mathcal{R},E_O}$
2. For a given condition τ and observation history $(o_0, \cdots, o_{\mathcal{R}})$ check whether:

$$A(o_0, \cdots, o_{\mathcal{R}}) = \forall x.\ Reach(x) \wedge (obs(x) = (o_0, \cdots, o_{\mathcal{R}})) \rightarrow \tau(x)$$

The second step of this procedure suggests us that we can synthesize the region of observations by quantifier elimination. In particular, by quantifying away the state variables x, in order to obtain an expression of A that only depends on the observable history $(o_0, \cdots, o_{\mathcal{R}})$.

This approach is guaranteed to terminate for finite systems. For example, we can use BDD-based techniques to compute the symbolic set of reachable states, and perform quantifier elimination. For infinite state systems, we cannot guarantee termination, since the computation of the reachables state might not be possible. Assuming that the reachables have been computed, the step of theory quantifier elimination is usually expensive, e.g., $\mathcal{LRA}$ [124], and sometimes undecidable, e.g., Arrays [43]. Therefore, depending on the theories used to model the system, we might be able (or not) to perform synthesis.

We will discuss the benefits and drawbacks of the up-front computation of the reachable states, in Chapter 11, when we will talk about temporal epistemic logic model checking. Indeed, the problem of synthesizing the alarm region is the same problem of understanding the set of states for which the epistemic formula $K\tau$ holds: i.e., $K\tau$ is a correct and maximal alarm. Recall that a correct and maximal alarm is defined as (e.g., for bounded delay):

$$G(\llcorner A \lrcorner \rightarrow O^{\leq d}\beta) \wedge G(\llcorner KO^{\leq d}\beta \lrcorner \rightarrow A)$$

A way to satisfy these constraints is to replace A with $KO^{\leq d}\beta$:

$$G(\llcorner KO^{\leq d}\beta \lrcorner \rightarrow O^{\leq d}\beta) \wedge G(\llcorner KO^{\leq d}\beta \lrcorner \rightarrow KO^{\leq d}\beta)$$

that is trivially satisfied by the knowledge axiom: $K\varphi \rightarrow \varphi$. We can do a similar reasoning for exact-delay and finite-delay. Computing the set of observations that satisfy $K\tau$ is equivalent to computing the observable denotation of $K\tau$.

8.2.2 Example

Let us consider an example with recall $\mathcal{R} = 0$, inspired by the Packet Utilization Service (Section 3.1.3). Our system has an internal reading x that is bounded between 0 and 5 during nominal operation. We introduce a (permanent) fault f, s.t., the value of x starts to steadily increase.

The SMV code for this example is shown in Figure 8.8. The set of reachables can be symbolically defined as:

$$Reach(x, f) \stackrel{\Delta}{=} (x \geq 0) \wedge (\neg f \rightarrow x \leq 5)$$

We now assume that the system provides as output an abstraction of the internal sensor, telling us whether the sensor is in range or not, i.e.,:

$$obs(x, f, o) \stackrel{\Delta}{=} o \leftrightarrow (x \geq 0 \wedge x \leq 5)$$

```
VAR x: real;
VAR f: boolean;
VAR o: boolean;

INIT x = 0;
INVAR !f -> (0 <= x & x <= 5)
TRANS (f -> next(f)) & (f -> (next(x) = x+1))
```

Figure 8.8: Example Plant (SMV Code)

we can thus proceed to synthesize the alarm for f:

$$A_f(o) \stackrel{\Delta}{=} \forall x, f.\ Reach(x) \wedge obs(x, f, o) \rightarrow f$$

$$A_f(o) \stackrel{\Delta}{=} \forall x, f.\ \big((x \geq 0) \wedge (\neg f \rightarrow x \leq 5) \wedge (o \leftrightarrow (x \geq 0 \wedge x \leq 5)) \rightarrow f\big)$$

we can quantify away the state variables x and f (e.g., by using MathSAT [53]) and obtain:

$$A_f(o) = \neg o$$

8.2.3 Magicbox

Let us consider the previous magicbox, and a diagnoser with recall 2. Recall 2 means that we can use the current observation together with the previous 2. This example is relatively small so that we can simulate what the synthesis algorithm would do. By looking at Figure 6.4, we can compute all traces of length 3. The task is simplified by the fact that any cell can be an initial state. From the set of traces, we compute the observations, expressed over the X and Y observers (Figure 8.9).

The alarm associated with the specification $\textsc{ExactDel}_K(A_{B1}, \beta_{B1}, 0, system, True)$ depends on the observations associated with the state `B1`, below we show which (and how many) observations are available to the diagnoser:

- Time 0: (x, y)

States			Observations		
t_0	t_1	t_2	t_0	t_1	t_2
A0	B1	A1	$(\overline{x}, \overline{y})$	(x, y)	$(\overline{x}, y)$
A1	B2	C1	$(\overline{x}, y)$	$(x, \overline{y})$	$(\overline{x}, y)$
A2	C0	A0	$(\overline{x}, \overline{y})$	$(\overline{x}, \overline{y})$	$(\overline{x}, \overline{y})$
A2	C0	B2	$(\overline{x}, \overline{y})$	$(\overline{x}, \overline{y})$	$(x, \overline{y})$
B0	A2	C0	$(x, \overline{y})$	$(\overline{x}, \overline{y})$	$(\overline{x}, \overline{y})$
B1	A1	B2	(x, y)	$(\overline{x}, y)$	$(x, \overline{y})$
B2	C1	B0	$(x, \overline{y})$	$(\overline{x}, y)$	$(x, \overline{y})$
C0	A0	B1	$(\overline{x}, \overline{y})$	$(\overline{x}, \overline{y})$	(x, y)
C0	B2	C1	$(\overline{x}, \overline{y})$	$(x, \overline{y})$	$(\overline{x}, y)$
C1	B0	A2	$(\overline{x}, y)$	$(x, \overline{y})$	$(\overline{x}, \overline{y})$

Figure 8.9: Traces and Observations for Recall 2

- Time 1: $(\overline{x}, \overline{y})$ (x, y)
- Time 2+: $(\overline{x}, \overline{y})$ $(\overline{x}, \overline{y})$ (x, y)

Initially, only the current state is used. Afterward, we slowly fill-up the memory of the diagnoser. Since the diagnoser has recall 2, any trace longer than two steps will have exactly 3 observations. We can see that even with recall 0, we can always distinguish the state `B1`, since it has the unique observation (x, y). Therefore, the alarm condition is system diagnosable for any recall $\mathcal{R} \geq 0$.

For $\textsc{ExactDel}_K(A_{C1}, \beta_{C1}, 0, trace, True)$ we have:

- Time 0: $(\overline{x}, y)$
- Time 1: $(x, \overline{y})$ $(\overline{x}, y)$
- Time 2+: $(\overline{x}, \overline{y})$ $(x, \overline{y})$ $(\overline{x}, y)$

we can see that with recall 0 we cannot distinguish between `C1` and `A1`. This tells us that with recall 0, the specification is not even trace diagnosable. Recall 1 is sufficient for trace diagnosability, since there is no ambiguity

in the observation. Notice that we cannot gain system diagnosability even if we increase the recall. This is because we will always have uncertainty in the initial states, since we only performed one observation. The Alarm A_{C1} is defined in terms of the two observations $o_1 = (x, \overline{y})$, $o_0 = (\overline{x}, y)$:

$$A_{C1}(x_0, y_0, x_1, y_1, x_2, y_2) = \neg x_0 \wedge y_0 \wedge x_1 \wedge \neg y_1$$

8.3 Experimental

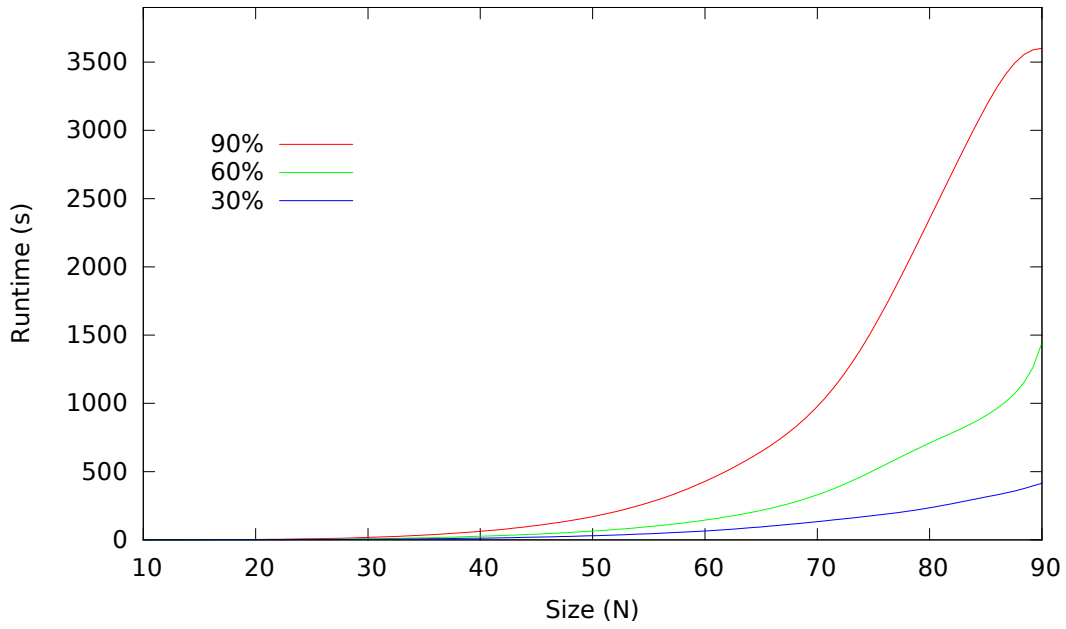

Figure 8.10: Perfect-Recall synthesis run-time with different observable percentages. (Unconstrained Init)

To understand the applicability of both approaches, we run an experimental evaluation on a set of finite state magicboxes[2]. In particular, we consider magicboxes varying in size from 10x10 to 90x90 (i.e., models with more than 8000 states). Since the magicboxes provide long paths, we consider two classes of problems. In the first class, any state can be an initial

[2]Tools and benchmarks are available at `http://marco.gario.org/phd/`

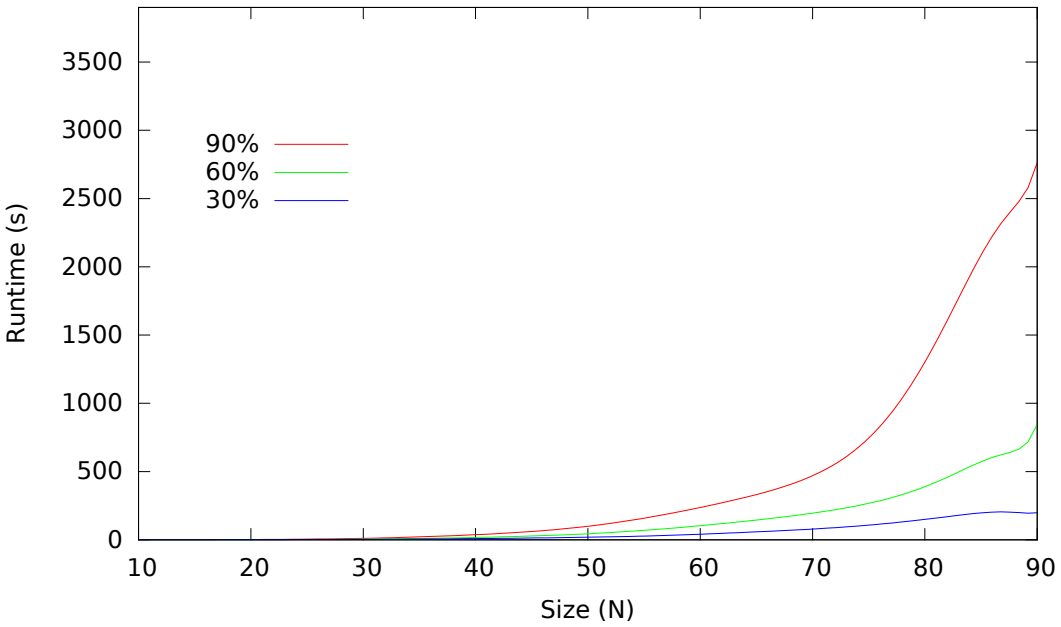

Figure 8.11: Perfect-Recall synthesis run-time with different observable percentages. (Constrained Init)

state. In the second class, we fix the column position of the initial states. Constraining the initial state makes counter-examples become deeper (i.e., longer counter-examples). On the other hand, it solves some uncertainty concerning the initial states. Finally, we also consider three different degrees of observability: 30%, 60% and 90%. This means that the given percentage of columns and rows are observable. The diagnosis condition that we verify is a 0-delay property, that requires knowing whether the ball is in a given cell.

Perfect Recall First of all, we study the result of the synthesis in the perfect recall case. In this case, we use a BDD-based engine, thus we expect the procedure to suffer from the increase of state variables (even when using techniques such as dynamic reordering). The other parameter that can have an impact on this is the number of observable variables. The more observable information is available, the more states will be created

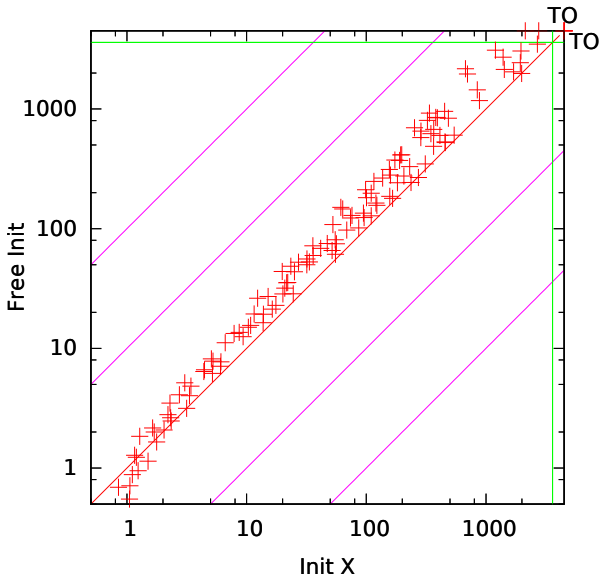

Figure 8.12: Perfect Recall: Effect of Init constraints on synthesis time.

by the belief explorer. This is true to a certain degree, since if everything is observable, then the belief explorer will coincide with the system itself.

Figures 8.10 and 8.11 show the runtime for the benchmarks with different degrees of observability, respectively when considering no constraint in the initial state, or constraining the initial column. The exponential impact of the size of the number of observables is clearly visible in both cases. Moreover, we can see that the synthesis algorithm is able to synthesize the diagnoser for magicboxes of size 90x90 in less than 10 minutes if we only have 30% observability, but with 90% we start running into time-outs. The plots are smoothed to show the general trend. By looking at the data, we can see that the actual runtime is impacted by both the dynamic reordering of the BDD and by the garbage collection operations, and it grows less smoothly. The impact of constraining the initial states is further demonstrated in Figure 8.12. By constraining the initial state, the belief explorer will have more information about the possible starting state (fewer initial belief states). In turn, this leads to less uncertainty and fewer belief states. The impact is constant, but not particularly significant.

Bounded Recall For the bounded recall case, we need to consider multiple sizes for the recall window. In particular, we consider recall 0 to 5. This brings us to a situation in which there are many more observable variables than non observable ones. For example, in a system with 100 variables and observability 30%, we have 30 observable variables per step. Thus, at recall 5 there are 150 variables that are used to recall the observation.

Figures 8.13 and 8.14 show the impact of increasing the observability, in the case of 0-recall, respectively, for the class without initial constraints and the one with initial constraints. It is immediately clear that the initial constraints, have a significant impact on this technique. In particular, we obtain counter-examples that are longer, and thus the synthesis engine needs to perform more work to build the region. Therefore, we notice that the depth of the counter-examples is a key factor for our synthesis procedure.

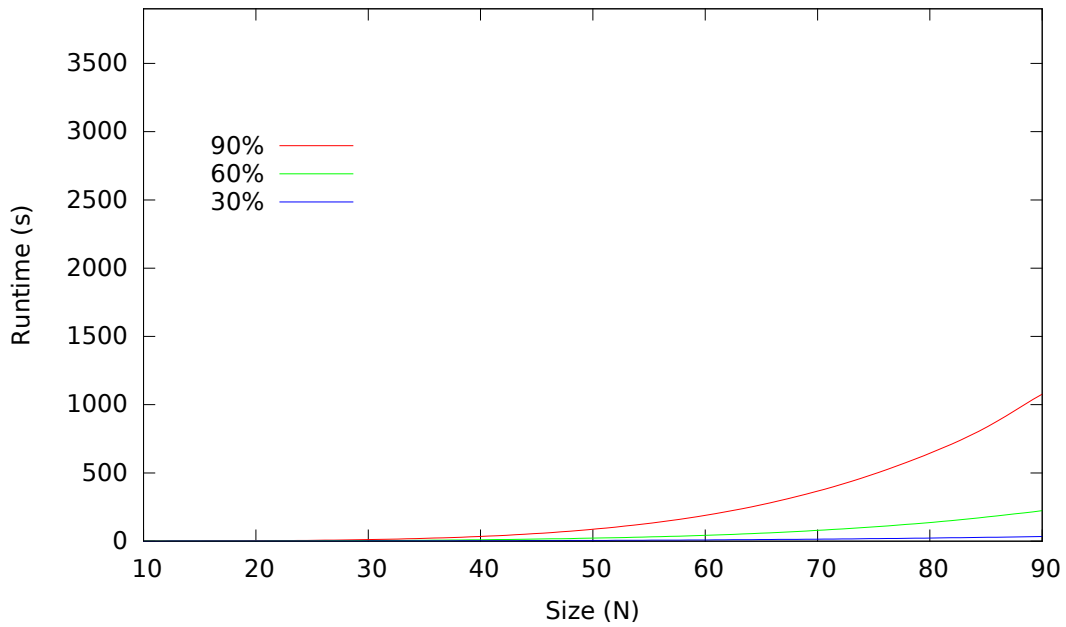

Figure 8.13: 0-Recall synthesis run-time with different observable percentages (unconstrained Init)

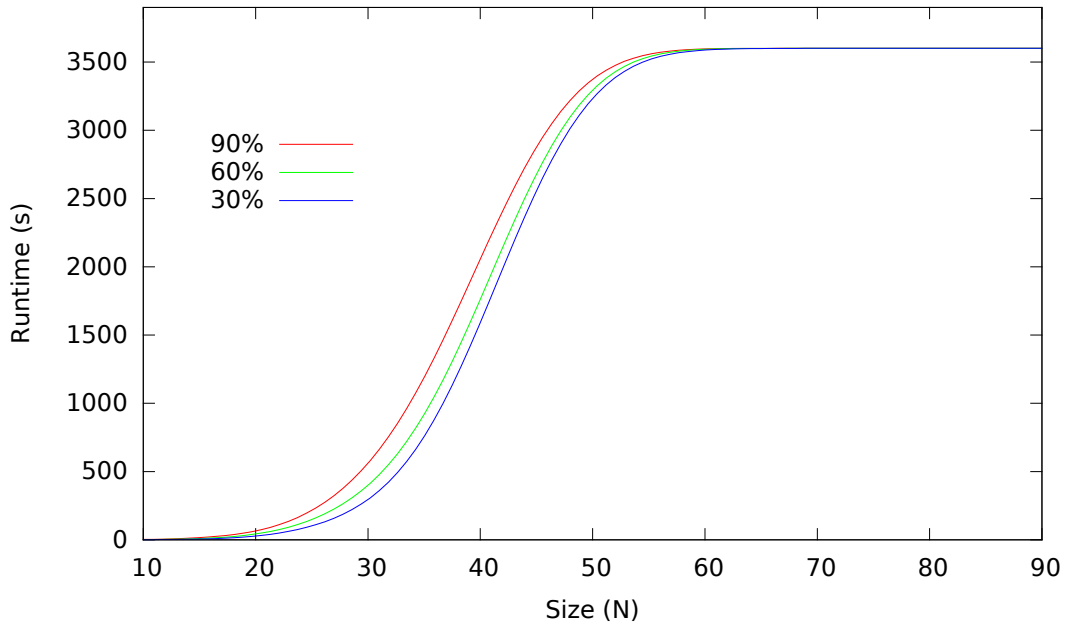

Figure 8.14: 0-Recall synthesis run-time with different observable percentages (constrained Init)

To study the impact of increasing the recall in the synthesis runtime, we only consider the class of problems without initial constraints and 30% of observability. Figure 8.15 tells us that going from recall 0 to recall 5 yields a significant increase in runtime. This suggests that in order to deal with long recall periods, we need to find a way to minimize the amount of information that we need to store. A possible idea is to exploit the sensor placement problem (Chapter 7.2) in order to understand which observations need to be stored in order to achieve diagnosability. In fact, it might turn out that a given observable is irrelevant and thus we do not need to store its value at all. This line of research is left for future work.

Comparison Finally, we compare the two techniques. When looking at the data, we need to take into account that the underlying technology for the two approaches is considerably different. Perfect recall synthesis is using BDDs, while bounded recall is using a SAT solver. In particu-

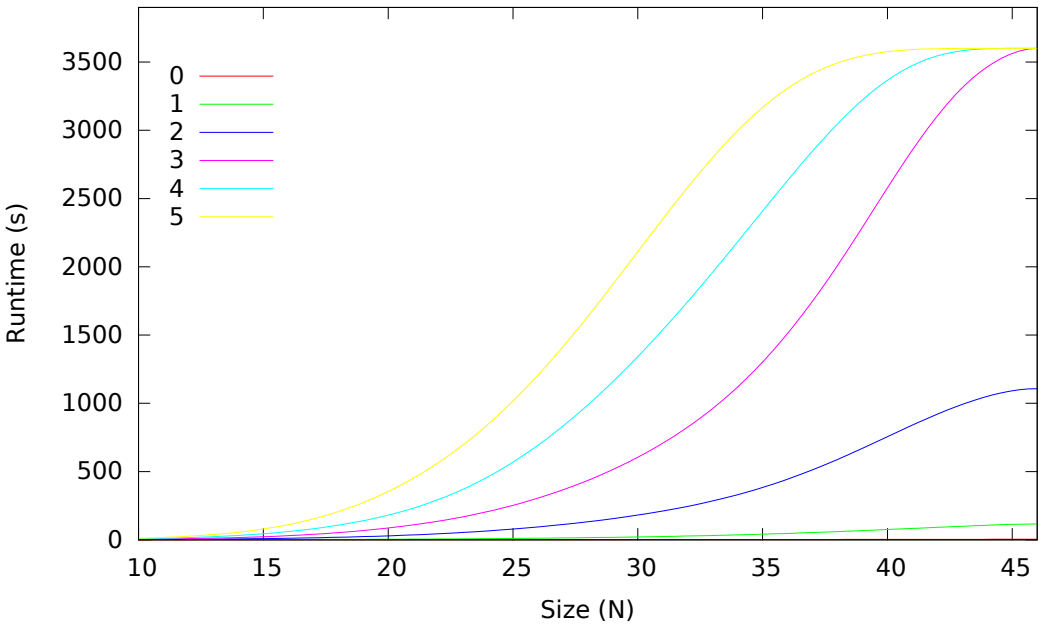

Figure 8.15: Synthesis run-time for different recall values (unconstrained Init – 30% Obs)

lar, Figures 8.16 the comparison between the two classes of models, and how in one case the perfect recall synthesis algorithm can outperform the bounded recall one (for recall 0). The red boxes show the class with unconstrained initial state, where the counter-examples are shallow, while the blue crosses represent the class with deep counter-examples. This shows that perfect recall not only provides more information than the bounded recall approach, but in some cases can even be synthesized faster.

8.4 Chapter Summary

Having a formal modal of the plant and formal specifications make it possible to perform automated synthesis of the FDI. In a sense, this is the ultimate step in our formal approach for FDI design. The main advantage of synthesis is the reduction in the design time. Being able to synthesize an FDI makes it possible to explore different strategies and designs with

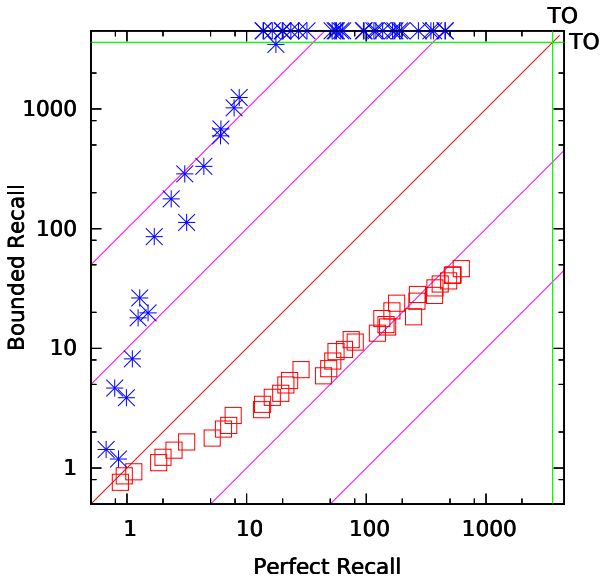

Figure 8.16: PR vs BR: Unconstrained Init (Red Boxes) - Constrained Init (Blue Crosses)

a short feedback loop. Moreover, changes to the plant or specification can be automatically addressed.

One of the drawbacks of synthesis is that it is hard for a human to understand the rationale behind the synthesized component, for example in all the settings where human certification is needed, or when modifications want to be introduced manually. In these situation, we still believe that the synthesis can play an important role, in order to guide the design. On one hand, it is possible to use synthesized components as placeholders while the manual FDI is being designed. This makes it possible to test/study the system together with the FDI, and explore other aspects (e.g., recovery). On the other hand, since the FDI is a deterministic component, the goodness of the FDI designed by hand can be certified by equivalence checking. In fact, all correct and maximal diagnosers for the same specification must have the same input-output behavior.

In this Chapter, we presented two techniques for automated synthesis. The main difference between the two approaches is the amount of recall of

the diagnoser, and thus the shape of the diagnoser. In the case of perfect recall, we build an automaton, in order to account for the growing history of observations. In the bounded recall case, we show that we can compactly represent the diagnoser as a combinational circuit.

The technique we presented for bounded recall work on both finite and infinite state systems. The techniques for perfect recall, however, cannot be applied directly to infinite state systems, and we leave their extension for future work. A simple way of achieving perfect recall on infinite state systems is to abstract the system and then apply finite state techniques. This approach will be discussed in the next Chapter, when considering Timed Failure Propagation Graphs.

Finally, we demonstrate both algorithms on several examples, and experimentally study their applicability. In particular, we show that to apply bounded recall for long windows, future work will have to identify ways to minimize the number of variables being introduced in the model.

Chapter 9

Timed Failure Propagation Graphs

Timely detection, identification and recovery of faults is an essential component for the correct operation of complex critical systems. Failures typically originate from basic faults, can manifest with delays, and can interact with each other over time in very complex ways. Timed Failure Propagation graphs (TFPGs) have been introduced in [105] as a framework to capture the temporal propagation of faults in complex systems, and to support important run-time activities such as diagnosis and prognosis. Intuitively, a TFPG is a graph-like model that accounts for the temporal progression of failures in dynamic systems and for the causality between failure effects, taking into consideration time delays, system reconfiguration, partial observability, and sensor failures.

In practice, TFPGs have been applied in several industrial contexts. In aerospace, NASA [96] positively evaluated them in the context of diagnostic technology for manned aircraft. Boeing has been using them [131] for performing maintenance monitoring of systems; in particular, in [6] they present an integrated vehicle health management system based on TFPGs. Furthermore, the European Space Agency has funded projects [76, 77] where TFPGs are used as the basis for the specification of fault detection and isolation components [25]. Recently, in addition to monitoring of

safety critical systems, there has been some interest in using TFPGs for performing software monitoring [72, 71]. A nice overview of applications of TFPGs is given in [145].

As in any model-based diagnosis approach, however, the quality of the diagnosis strongly depends on the quality of the model. Unfortunately, the practical application of TFPGs is currently clashing against the lack of suitable validation methods and tools. TFPGs are usually built manually by safety engineers, based on their own knowledge of the relations between faults in the system, and based on safety artifacts such as Fault-Trees [156]. In the few cases in which they can be generated from a system model, there is no way for the users to modify or adapt them and still be able to certify their quality.

To be able to use the TFPG as an abstraction of the system to diagnose, the TFPG needs to capture all interesting behaviors of the system. In Section 9.1 we present a validation approach based on Satisfiability Modulo Theory (SMT). This makes it possible to study the behavior of the TFPG, perform diagnosability analysis and use the TFPG as a model for online diagnosis. We present several reasoning problems, and provide experimental data to demonstrate the feasibility of the approach.

In Section 9.2, we show how to use the TFPG as an abstraction of the system, and discuss techniques for performing diagnosability analysis on an abstraction of the system. In order to do so, we first introduce a transformation technique that reduces a TFPG into a discrete-time transition system. This technique allows us to apply the techniques discussed in the previous chapter for discrete time systems on timed systems, and it is at the base of the FAME project (Chapter 10). Afterwards, we discuss the possibility of using a diagnoser for the TFPG for the original plant (diagnoser reuse), by introducing the concept of cross-diagnosability.

The contributions of this Chapter are:

- A symbolic technique for validation of TFPG based on SMT (originally presented in [31]);
- A translation from TFPG to Transition System (in collaboration with Benjamin Bittner);
- Cross-Diagnosability approach for reasoning about diagnoser reuse and abstractions of the plant.

9.1 TFPG Validation

TFPGs are usually validated via extensive testing, and no approach exists for performing a more exhaustive analysis and validation of the model. However, since TFPGs are used as a basis for diagnosis, it is fundamental to be able to establish their formal properties, and to gain confidence in their adequacy — similarly to other model-based approaches, the effectiveness of the reasoning relies on the accuracy of the model. Interest in validation is witnessed in [147], that introduces a data-driven method called *alarm-sequence maturation* in order to correct errors in the causality relations of the model. This method, however, requires data to be already available, and thus can be applied only after the overall system has been put into production. In many contexts, e.g., aerospace, it is not possible or desirable to wait until deployment of the system to validate the TFPG. Thus preliminary validation should be performed in the design phase.

In this chapter, we propose a practical approach to support the analysis and validation of TFPG models. We define some important design-time queries, and show how they can be efficiently answered by performing symbolic reasoning. Our approach is based on a logical characterization of TFPGs, and cast in the field of Satisfiability Modulo Theory (SMT) [12]. The set of possible executions of a TFPG is modeled as a formula in SMT.

The reasoning tasks are expressed in form of logical queries in SMT, and the implementation is based on the use of an efficient SMT solver. We explore the problems of *validation*, *refinement testing*, and *diagnosis and diagnosability*. The approach has been implemented and evaluated experimentally, and it is able to leverage symbolic SMT-based techniques to deal with the state-space explosion problem. In this work, we focus only on three possible analyses, but using an SMT characterization of the TFPG, it becomes possible to easily define new analyses, or implement other more classical problems, such as prognosis.

Our approach is complementary to the available tools, such as the state-of-the-art FACT tool-set [104], where limited support for model validation is provided. In FACT, for example, it is not possible to provide a condition on a TFPG and obtain a complete execution satisfying it automatically, nor verify properties against the TFPG. One could use our approach to test the correctness of the TFPG model, and then perform diagnosis and prognosis using the FACT tool-set.

9.1.1 Satisfiability Modulo Theory

This work on validation of TFPGs leverages the impressive advancement in Satisfiability Modulo Theory (SMT) (Section 2.1). We use the theory of linear arithmetic over the real numbers ($\mathcal{LRA}$). Difference logic ($\mathcal{RDL}$) is the subset of $\mathcal{LRA}$ such that atoms have the form $x_i - x_j \bowtie c$. We write $x - y \in [a, b]$ meaning the formula $(x - y) \geq a \wedge (x - y) \leq b$. If a is $-\infty$ then the first conjunct is omitted and similarly, if b is $+\infty$ then the second conjunct is omitted.

9.1.2 Timed Failure Propagation Graphs

The Battery Sensor System (BSS) (Figure 9.1) will be our running example. The example is explained in detail in Section 4.5, here we recall the main ideas. The BSS provides a redundant reading of the sensors to a device.

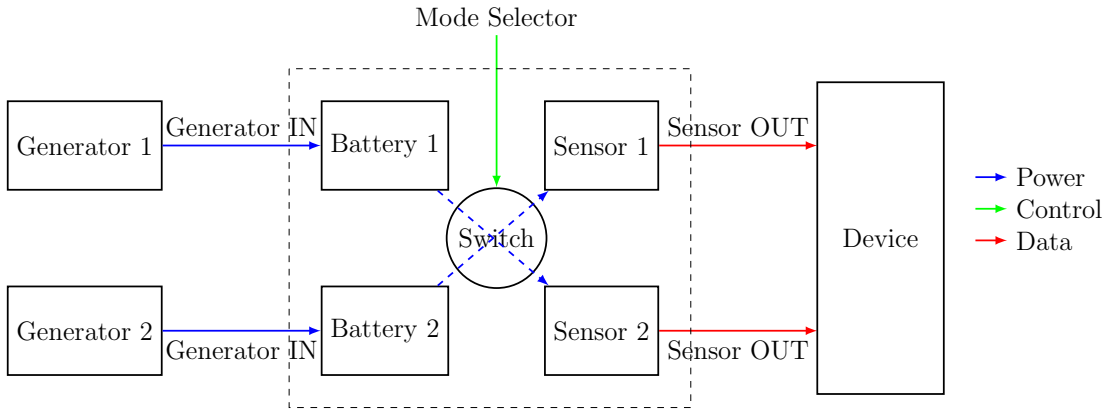

Figure 9.1: Battery Sensor System schema

Internal batteries provide backup in case of failure of the power supply. The safety of the system depends on at least one of the sensors providing a reading at any time.

A *failure mode* is a failure of a component of the system. A component might fail in more than one way, therefore having more than one failure mode associated with it. We call *fault* the occurrence of a failure of the component. A fault in a component will produce anomalies in the system behavior, that we call *discrepancies*. In the BSS, we define one failure mode for the generator, and one for the sensor. The generator can fail and stop supplying power (G_{Off}), and the sensor can stop providing a reading (S_{Off}). After the generator fails, the battery will start discharging. When the battery is exhausted, the attached sensors will stop working. Examples of discrepancies are:

- Absence of power from the generator,
- Battery level going below a threshold,

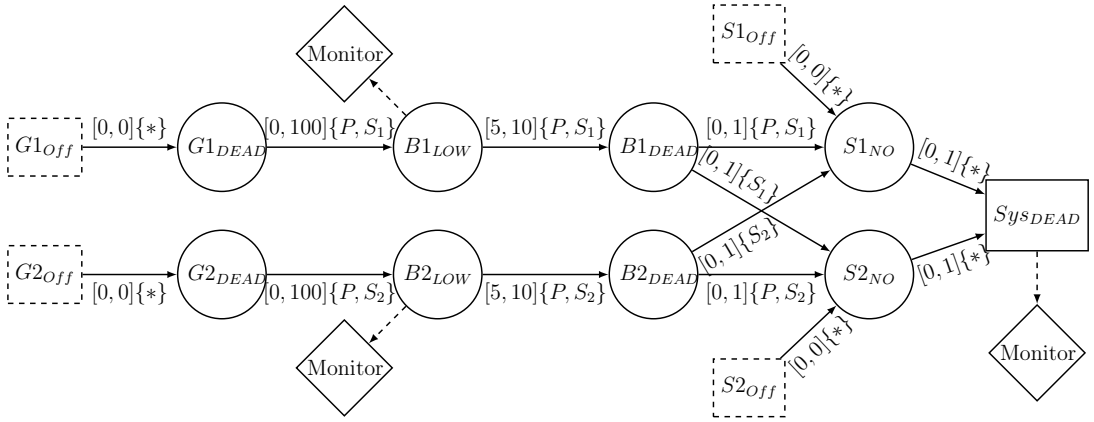

Figure 9.2: TFPG for the BSS example."$*$" is used to indicate all modes: $\{P, S_1, S_2\}$

- Sensor not providing a reading.

Some discrepancies have a monitor attached (i.e., a sensor), and we call them *monitored discrepancies*. In our example, we monitor the level of the battery, and are warned when the level goes below a certain threshold. Not all discrepancies can be monitored, due to physical or design limitations. Intuitively, we can monitor all those information that can be captured and analyzed with sensors. The semantics of monitored discrepancies makes it possible for monitors to fail. In this case, the monitor can provide both false positive and false negative information.

System modes (simply *modes*) are configurations of the system that are relevant for capturing the propagation of faults. In the BSS the sensors are powered by their own battery, however, in case of faults, one battery can power both. We define 3 modes: $Primary$, $Secondary_1$, $Secondary_2$. If Generator 1 fails while the system is powered by Generator 2, there will be no impact on the system.

Timed Failure Propagation Graphs (TFPG) were introduced in [105] to model the progression of faults in a system and perform online diagnosis and prognosis. A TFPG is a directed graph model where nodes represent *failure modes* and *discrepancies*. Edges represent the causality

between nodes, and provide information on the delay in the propagation. By labeling the edges with *system modes*, we can encode switching-systems in which different propagations are possible in the different modes. Figure 9.2 shows a TFPG for the BSS. Boxes with dotted lines are failure modes, whereas discrepancies are either circles (`OR` discrepancies) or boxes (`AND` discrepancies) with solid lines. $B1_{LOW}$, $B2_{LOW}$ and Sys_{DEAD} are monitored discrepancies, thus we show the monitor graphically (with diamonds). Edges include information on the propagation time and modes in which they are active.

The non-determinism on the propagation time between $G1_{DEAD}$ and $B1_{LOW}$ models the fact that we do not know the charge level of the battery until we get to a critical level (B_{LOW}). Additionally, we allow for a small non-determinism between the activation of the discrepancy, the absence of output ($S1_{NO}$,$S2_{NO}$) and the failure of the system, in order to model the possibility of the device to handle incorrect sensor readings. The uncertainty on the propagation time between B_{LOW} and B_{DEAD} is motivated by the fact that the depletion of the battery will take more time if we are in the *Primary* mode rather than in the $Secondary_1$ (or $Secondary_2$) mode (see charge update rule in Section 4.5).

Definition 28. *TFPG. A TFPG is a structure* $G = \langle F, D, E, M, ET, EM, DC, DS \rangle$, *where* F *is a non-empty set of failure modes;* D *is a non-empty set of discrepancies;* $E \subseteq V \times V$ *is a set of edges connecting the nodes* $V = F \cup D$; M *is a non-empty set of system modes. At each time instant the system can be in only 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 min/max propagation time on the edge (where,* $I \in \mathbb{R}_+ \times (\mathbb{R}_+ \cup \{+\infty\})$ *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 any edge* $e \in E$. $DC : D \to \{\texttt{AND}, \texttt{OR}\}$ *is a map defining the*

type; $DS : D \rightarrow \{\texttt{M}, \texttt{I}\}$ *defines whether the discrepancy is monitored (*`M`*) or not (*`I` *– inactive).*

Failure modes are always root nodes, and all discrepancies must have at least one incoming edge.

The semantics [4] of TFPGs says that the state of a node indicates whether the failure effects reached that node. A failure propagates through an edge $e = (v, w)$, only if the edge is active throughout the propagation, that is, from the moment v activates to the time w activates. An edge e is active if and only if the current mode m of the system is compatible with the modes of the edge ($m \in EM(e)$). For an `OR` node w and an edge $e = (v, w) \in E$, once a failure effect reaches v at time t, it will reach w at a time t', where $e.tmin \leq t' - t \leq e.tmax$ and the edge e is active during the whole propagation. On the other hand, the activation period of an `AND` node v' is the composition of the activation periods for each link $(v, w) \in E$. If an edge is deactivated any time during the propagation (due to mode switching), the propagation stops. Links are assumed memoryless thus a failure propagation is independent of any (incomplete) previous propagation. Finally, once a node activates, it changes permanently, and will not be affected by any future failure propagation.

In this chapter, we restrict ourselves to TFPGs that satisfy the following two conditions:

1. Cycles are strongly causal

2. The mode is frozen

Given a cyclic path in the TFPG, we say that it is *strongly causal* iff the sum of all t_{min} (of the edges in the path) is greater than zero. In other words, the cyclic path cannot propagate instantly. This is usually the case in physical systems, in which failure takes time to propagate. On the

contrary, when modeling logical systems, the propagations might capture a relation between discrepancies, and thus require to be instantaneous.

The *frozen mode* assumption requires that the mode of the TFPG is constants, i.e., we fix any of the available modes beforehand, and we assume that it does not change for the entire execution of the system. This assumption has been validated by domain experts, and it is justified by the fact that most of the reasoning (e.g., diagnosis) is of interest when the system is in a stable state. Given the number of system changes and possibly unpredictable interactions, modeling of fault propagation during mode-switching is not considered in practice.

9.1.3 TFPG as an SMT formula

A TFPG can be encoded into an SMT formula using the $\mathcal{RDL}$ theory. The encoding closely follows the definition of TFPG, and has been extensively validated using existing case-studies, and randomly generated TFPGs.

The encoding is divided into two parts. The first part, captures the behavior of the TFPG, and captures all valid executions of the TFPG. The formula capturing this part is called $ftfpg$. The second part, deals with the observability issue, and allows us to distinguish between monitored and unmonitored discrepancies. This part (the observable TFPG – $otfpg$) is built using $ftfpg$ as a building block.

$ftfpg$. Each node in the TFPG is associated to a *state variable* and an *activation time-point*. The semantics of the TFPG is encoded by defining constraints for the `OR` and `AND` nodes.

Given a TFPG G we create a formula $ftfpg(\vec{ud}, \vec{udt}, m)$, where $\vec{ud}$ and $\vec{udt}$ are vectors that define, respectively, the state (active or not) and the activation time of each node in G, and m is the current system mode. $\vec{ud}$ and $\vec{udt}$ are *unobservable* information and for brevity we combine them in

a single vector $\vec{u} = \vec{ud} \cup \vec{udt}$. Let us define $M(e, m) = \bigvee_{\mu \in EM(e)}(m = \mu)$ as an expression defining whether an edge e is compatible with the mode m; $ftfpg$ is defined as:

$$ftfpg(\vec{u}, m) = \bigwedge_{v \in D.\ DC(v)=\texttt{OR}} B_{or}(v, m)\ \wedge\ T_{or}(v, m)\ \wedge \bigwedge_{v \in D.\ DC(v)=\texttt{AND}} B_{and}(v, m)\ \wedge T_{and}(v, m)$$

$$B_{or}(v, m) = \vec{ud}(v) \leftrightarrow \bigvee_{(w,v) \in E} [\vec{ud}(w)\ \wedge M((w, v), m)]$$

$$B_{and}(v, m) = \vec{ud}(v) \leftrightarrow \bigwedge_{(w,v) \in E} [\vec{ud}(w)\ \wedge M((w, v), m)]$$

$$T_{or}(v, m) = \vec{ud}(v) \rightarrow [\bigvee_{(w,v) \in E} \Big(\vec{ud}(w)\ \wedge\ (\vec{udt}(v) - \vec{udt}(w)) \in ET((w, v))\Big)\ \wedge \bigwedge_{(w,v) \in E} \Big(\vec{ud}(w) \rightarrow\ (\vec{udt}(v) - \vec{udt}(w)) \leq t_{max}((w, v))\Big)\ ]$$

$$T_{and}(v, m) = \vec{ud}(v) \rightarrow [\bigwedge_{(w,v) \in E} \Big(\vec{ud}(w)\ \wedge (\vec{udt}(v) - \vec{udt}(w)) \geq t_{min}((w, v))\Big)\ \wedge \bigvee_{(w,v) \in E} \Big(\vec{ud}(v) \wedge\ (\vec{udt}(v) - \vec{udt}(w)) \leq t_{max}((w, v))\Big)\ ]$$

The encoding is composed of a Boolean part (B_{or}, B_{and}), that captures the structure of the TFPG, and of a Temporal part (T_{or}, T_{and}), that captures the temporal relation between discrepancies. Intuitively, the Boolean part expresses the possible combinations of nodes that can be active, while

the temporal part encodes the temporal relation between their activation. For each `OR` and `AND`, we express constraints on both the Boolean and temporal part.

In B_{or} we state that a node is active if at least one of the predecessors is active and the mode is compatible. In B_{and} we require all predecessors to be active. T_{or} states that given an active node, there must be at least one predecessor with an activation time that is compatible with the t_{min}/t_{max}. We additionally require that all *active* predecessors nodes have an activation time that is compatible with the t_{max}. Finally, T_{and} encodes that if a node is active, all the predecessors are active with timings satisfying the t_{min} constraint, and that at least one satisfies the t_{max}. Note that the second part of T_{and} is similar to that of T_{or}. However, the key difference is that in the `AND` case, we allow all but one active discrepancies to violate the t_{max} constraint. The failure modes are left unconstrained.

A model for $ftfpg$ represents a possible execution of the TFPG. This encoding is polynomial in the number of nodes in the TFPG. In particular, it uses $O(|E|)$ theory ($\mathcal{RDL}$) atoms.

otfpg Diagnosis requires the ability to reason on the monitored (i.e., observable) discrepancies. In order to do so, we extend the encoding by adding new variables for the monitors. The monitor has the same state and activation time as the discrepancy, however, in order to be able to consider monitoring faults, these constraints are conditioned to a set of health variables $\vec{h}$. $\vec{o} = \vec{od} \cup \vec{odt}$ are the *observable* variables, and $otfpg(\vec{o}, \vec{u}, m, \vec{h})$ is the TFPG with monitors:

$$otfpg(\vec{o}, \vec{u}, m, \vec{h}) = ftfpg(\vec{u}, m) \wedge \bigwedge_{v \in D.DS(v)=\texttt{M}} \vec{h}(v) \rightarrow (\vec{od}(v) = \vec{ud}(v) \wedge \vec{odt}(v) = \vec{udt}(v))$$

9.1.4 SMT-based Reasoning

The SMT encoding can be used to perform reasoning on the TFPG. We distinguish between reasoning tasks that can be done at *design time* or *runtime* (i.e., when a system is running). The objective of design time reasoning is to help the designer validate the TFPG, i.e., show that the TFPG captures the situations of interest. We focus on design time analysis, but also provide an example of how to perform a runtime task (diagnosis).

The design time reasoning task that we consider are the following: *model validation*, *refinement testing* and *diagnosability*. All these problems can be handled in a uniform way by leveraging the SMT solver, without the need of defining multiple ad-hoc algorithms.

Partial Traces The SMT encoding of the TFPG represents the set of all possible executions (i.e., traces) of the TFPG: every model of the formula is a trace of the TFPG (and vice versa). A trace, in this context, is a complete assignment to the activation state and time variables, i.e., whether and when the discrepancy activated.

The SMT solver can provide provide us with a trace of the TFPG, that also satisfies additional constraints on how this trace should look like. E.g., we can ask for a trace of the BSS in which the discrepancy $B2_{LOW}$ is active by using the formula $\tau(\vec{u}, m) = ud(B2_{LOW})$. Such a trace can be obtained from the SMT solver by asking for a model of:

$$ftfpg(\vec{u}, m) \wedge \tau(\vec{u}, m)$$

A *model* is an assignment of concrete values to all the $\vec{ud}$, $\vec{udt}$ and m variables, telling which discrepancies and failure modes are active, and their activation time. If no model exists (i.e., formula is unsatisfiable) there is no trace in the TFPG that satisfies the given requirement. A *partial trace* can be defined by describing the activation of some discrepancies without the

need of specifying the behavior of all nodes. The SMT solver will provide a complete trace that satisfies the constraint τ, thus enabling the user to explore the behavior of the TFPG. Any $\mathcal{LRA}$ formula over the mode, state and time-point variables of the TFPG can be used to perform querying (in place of τ). This provides a wide degree of expressiveness.

Model Validation Building on the idea of partial traces, we can ask whether some behavior is possible or not in the TFPG. Knowledge of the domain and of the original model can be used to define properties that we expect the TFPG to satisfy. In the BSS, the failure of the generator $G1$ *may* lead $S1$ to stop working. However, if the system is in mode $Secondary_2$ the failure of $G1$ will not have any impact on $S1$. We call this type of property a *possibility*. We are interested in checking that some behaviors are possible in the TFPG. To do so, we use the concept of partial traces, and ask whether there exists a trace in the TFPG that satisfies our requirement. This is done by checking the satifiability of:

$$ftfpg(\vec{u}, m) \wedge \vec{ud}(G1_{Off}) \wedge \vec{ud}(S1_{NO})$$

An example of possibility checking is whether all nodes of the TFPG can eventually be activated. E.g., it might be that not all nodes can be activated in a given mode. A node that cannot be activated in any mode represents a modeling error. Such nodes can be found by running multiple possibility checks, optionally specifying in which modes we expect the node to be enabled. E.g., we cannot activate the discrepancy $B1_{LOW}$ in the mode $Secondary_2$, thus the following is unsatisfiable:

$$ftfpg(\vec{u}, m) \wedge \vec{ud}(B1_{LOW}) \wedge m = Secondary_2$$

The counter-part of possibility is *necessity*. In the BPSS we know that a battery cannot discharge if the associated generator is working. This

property can be specified as follows. For $B1_{LOW}$ to be active, it is necessary for $G1_{Off}$ to be active. We check the validity of:

$$ftfpg(\vec{u}, m) \rightarrow \left(\vec{ud}(B1_{LOW}) \rightarrow \vec{ud}(G1_{Off})\right)$$

In words, every trace of the TFPG in which $B1_{LOW}$ is active requires $G1_{Off}$ to be active.

The purpose of model validation is to increase the confidence on the correct behavior of the model. Let us introduce an artificial bug, and modify the TFPG of the BSS so that the mode on the edge $(B2_{DEAD}, S1_{NO})$ becomes $Secondary_1$. This implies that there is no propagation between those two nodes in mode $Secondary_2$. This mistake could be detected with the following possibility query: *It is possible for the single failure mode* $G2_{Off}$ *to cause* Sys_{DEAD}. Therefore, we want to find a trace in which $G2_{Off}$ and Sys_{DEAD} are active, but all other fault are not active.

$$ftfpg(\vec{u}, m) \wedge \neg(\vec{ud}(G1_{Off}) \vee \vec{ud}(S1_{Off}) \vee \vec{ud}(S2_{Off})) \\ \wedge \vec{ud}(G2_{Off}) \wedge \vec{ud}(Sys_{DEAD})$$

This execution is possible in the original TFPG (in the $Secondary_2$ mode) but not in the modified one.

Refinement Changes in the models are a common activity during development. Thus, it is important to show some relation between the original model and the modified one. In the BSS example, there is uncertainty on the propagation time between $B1_{LOW}$ and $B1_{DEAD}$ due to the different discharge rates of the battery in the primary and secondary mode. This uncertainty can be removed by adding two intermediate discrepancies ($B1_P$, $B1_S$) that have an incoming edge from $B1_{LOW}$ and an outgoing edge to $B1_{DEAD}$. Intuitively, $B1_P$ can be reached only in mode $Primary$, thus providing an exact value for the propagation ($ET = [10, 10]$) (similarly for

$B1_S$ in $Secondary_1$ with $ET = [5, 5]$). We need to check that this new TFPG (BSS_2) refines the original TFPG (BSS_1). Therefore, we define a mapping between the two saying that all nodes that exist in both TFPGs must have the same state and activation times, and check that each execution in the new TFPG (BSS_2) has a corresponding execution in the old one (BSS_1), i.e., it is a refinement.

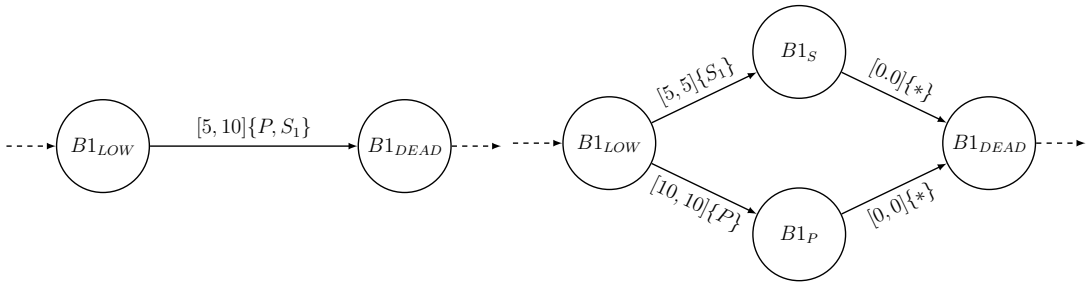

Figure 9.3: TFPG Refinement

An example of mapping is defined by the following relation:

$$\gamma(\vec{u}_1, \vec{u}_2) = \bigwedge_{v \in V} (\vec{ud}_1(v) \leftrightarrow \vec{ud}_2(v)) \wedge (\vec{ud}_2(v) \rightarrow \vec{udt}_1(v) = \vec{udt}_2(v)) \tag{9.1}$$

Given two TFPGs G_1, G_2 and a (partial) mapping $\gamma(\vec{u}_1, \vec{u}_2)$ between their nodes, we say that G_1 refines G_2 if every trace of G_1 can be mapped to a trace of G_2:

$$\forall \vec{u}_1, m. ftfpg_{G1}(\vec{u}_1, m) \rightarrow \exists \vec{u}_2. (\gamma(\vec{u}_1, \vec{u}_2) \wedge ftfpg_{G2}(\vec{u}_2, m))$$

In practice, we use the negation of the formula above. In this way, we can obtain a counter-example to the refinement, i.e., a trace that belongs to the first TFPG but that cannot be mapped to any trace of the second:

$$ftfpg_{G1}(\vec{u}_1, m) \wedge \forall \vec{u}_2. \neg(\gamma(\vec{u}_1, \vec{u}_2) \wedge ftfpg_{G2}(\vec{u}_2, m))$$

Having a concrete counter-example is useful for debugging.

Diagnosis and Diagnosability In the context of TFPG, the goal of *diagnosis* is to understand which failure modes caused the observed discrepancies. This is achieved by generating all possible executions consistent with the observations and considering all the sets of faults that occur in those executions (Model Based Diagnosis [134]).

The concept of diagnosis condition (Section 5.1) can be generalized to the setting of TFPGs. A *diagnosis condition* $\beta(\vec{u})$ is a relation on the unobservable discrepancies and time points. In the most basic case, a diagnosis condition is a single failure mode: $\beta(\vec{u}) = \vec{ud}(fm)$. In the general case, a state condition captures a situation of interest in the system. E.g., the fact that a failure mode occurred in a given time-frame: $\beta(\vec{u}) = \vec{ud}(fm) \wedge 5 \leq \vec{udt}(fm) \leq 10$ or that two discrepancies activated in a particular order:

$$\beta(\vec{u}) = \vec{ud}(D1) \wedge \vec{ud}(D2) \wedge \vec{udt}(D1) \leq \vec{udt}(D2)$$

To perform diagnosis, we ask if a given diagnosis condition is possible given the observations provided by the monitored discrepancies. This basically boils down to a possibility check in which we define $\tau = \beta(\vec{u}) \wedge obs(\vec{o})$, where $obs(\vec{o})$ denotes the *observation*. In the setting of FDI alarms (Section 5.2) we require the diagnoser to be sure about the occurrence of a diagnosis condition. Therefore, we require that there is only one possible explanation for the given observation. Thus, instead of checking possibility, we check necessity. We can ask whether given the observable trace $obs(\vec{o}) = \vec{od}(Sys_{DEAD}) \wedge \vec{odt}(Sys_{DEAD}) = 10$ (Sys_{DEAD} was activated at time $t = 10$) it is necessary for $S1_{Off}$ to have been activated before time $t = 10$: $\beta(\vec{u}) = \vec{ud}(S1_{Off}) \wedge \vec{udt}(S1_{Off}) \leq 10$. If $otfpg(\vec{o}, \vec{u}, m, \vec{h}) \wedge obs(\vec{o}) \wedge \neg\beta(\vec{u})$ is unsatisfiable, the diagnosis condition is the only possible explanation for the observation, otherwise we are provided with a counter-example. The observable trace *obs* can be any $\mathcal{LRA}$

formula, thus we can express complex patterns of observations, e.g., temporal uncertainty: $obs = 5 \leq \vec{odt}(Sys_{DEAD}) \leq 10$.

At design time we want to have some guarantee on the effectiveness of the diagnoser at runtime. This can be achieved by performing *diagnosability* analysis. We generalize the twin-plant construction [101] (see Chapter 7), in order to find a critical-pair, i.e., a pair of traces that have the same observations s.t. one satisfies the condition but the other does not. The existence of a critical pair is tested with the following query:

$$otfpg(\vec{o}, \vec{u}_1, m, \vec{h}_1) \wedge otfpg(\vec{o}, \vec{u}_2, m, \vec{h}_2) \wedge \beta(\vec{u}_1) \wedge \neg\beta(\vec{u}_2) \wedge Healthy(\vec{h}_1, \vec{h}_2) \tag{9.2}$$

notice, that we enforce some relation on the health variables of the monitors. Usually, we are interested in checking diagnosability when all sensors are working correctly. However, we can explore other configurations for the health variables. For example, we can explore how many sensor failures we can afford before a given diagnosis condition becomes non-diagnosable by changing the relation $Healthy$: e.g., require that at most 2 monitors can fail.

This basic approach to diagnosability requires that all observations have been made. Due to the importance of timing in detection and recovery of faults, it might not be possible to wait until all observations have been made. In particular, we want to know which observations are sufficient to distinguish a faulty trace. This can be checked by performing the diagnosability analysis for subsets of observables. The observable variables are partitioned in two sets: $\vec{o}_A$, $\vec{o}_B$. Intuitively, we ask that the two systems are observationally equivalent only for a given subset of the observable discrepancies ($\vec{o}_A$). Thus, the diagnosability test becomes:

$$otfpg(\vec{o}_A \cup \vec{o}_{B1}, \vec{u}_1, m, \vec{h}_1) \wedge otfpg(\vec{o}_A \cup \vec{o}_{B2}, \vec{u}_2, m, \vec{h}_2) \wedge \beta(\vec{u}_1) \wedge \neg\beta(\vec{u}_2) \wedge Healthy(\vec{h}_1, \vec{h}_2)$$

If no critical pair exists in this case, we do not need to wait for the other observation to understand whether the state condition occurred. This technique can also be used to search for sets of sensors that are sufficient to guarantee diagnosability [26] (Chapter 7). Since the above formula is unsatisfiabile if there is no critical pair, unsat-core extraction techniques can be used to minimize the number of used observable variables.

9.1.5 Experimental Evaluation

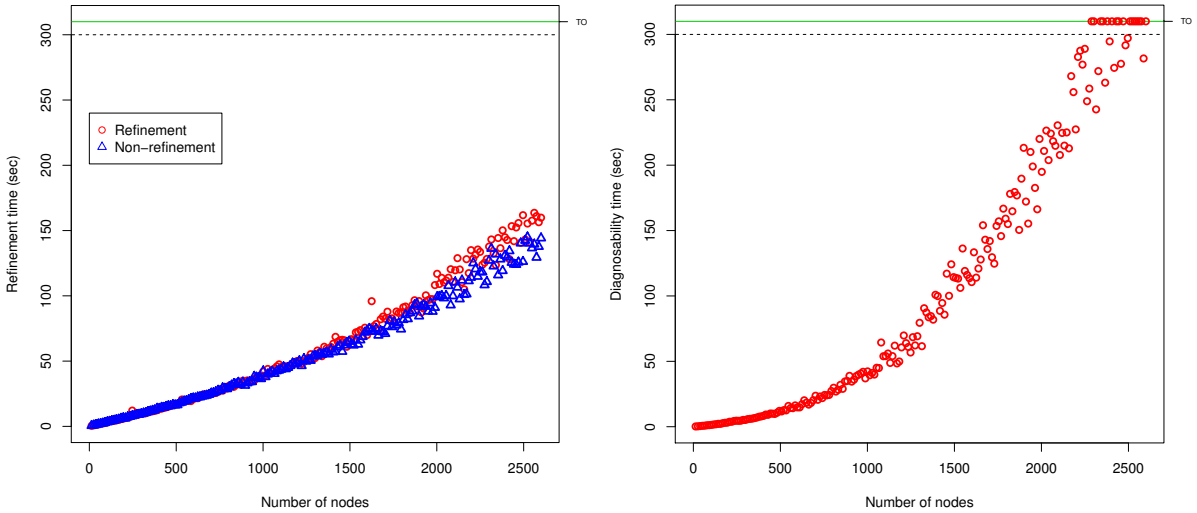

Figure 9.4: Scalability of the Refinement and Diagnosability. The "TO" line marks the examples that reached the timeout.

The techniques described in this paper have been implemented within the xSAP tool-set [24, 158], by relying on the pySMT [89] library for SMT formulae manipulation and integration with SMT solvers. The tool is able to generate partial traces for a TFPG, test possibility and necessity of arbitrary conditions, check refinement, and perform diagnosis and diagnosability. Due to the lack of publicly available TFPGs, we evaluated the scalability of the approach on a benchmark of randomly generated TFPGs [1]

Our approach has been evaluated on two sets of experiments: refinement and diagnosability. For the refinement benchmarks we take a TFPG and

[1] A dedicated version of the tool and benchmarks are available at `http://marco.gario.org/phd/`

derive a positive and a negative refinement instance. The solver is asked to verify whether those instances are refinements of the original TFPG. The transformation for the positive case requires picking a node of the TFPG, removing it and reconnecting the predecessors with the successors in a suitable way. Negative examples are obtained in a similar way, but we modify the propagation intervals to make them incompatible with the original ones. Since the check of refinement involves a quantifier alternation, we want to minimize the number of variables, and thus optimize the formula by inlining whenever possible. For the diagnosability problem, we test diagnosability of each failure mode in the TFPG, assuming that all sensors are healthy. We run our experiments using Z3 [66] as SMT Solver on an Intel i7 2.93GHz, using a time-out of 300 seconds and memory-out of 2GB for all experiments. The left plot of Figure 9.4 shows the runtime of the refinement testing when we increase the size of the TFPG, while the right plot shows the runtime for the diagnosability testing. The runtime of the diagnosability check increases quickly, since we are using the twin-plant construction: for every additional node in the TFPG we add four new variables to the problem.

The industrial TFPGs, to which we have access, are trivially analyzed by our approach. In the literature [96], TFPGs with 400 nodes are considered medium size, having more than 1000 nodes is uncommon. Therefore, our benchmarks consider examples that are reasonably bigger than commonly developed TFPGs. These experiments show that we are able to analyze huge TFPGs in a reasonable time; hence, we believe that these techniques could be integrated in the design process loop, providing quick feedback to the designer. The choice of using the number of nodes of the TFPG as indicator of its complexity is justified by previous work [3] where the algorithmic complexity of the reasoning algorithm is defined in terms of the number of nodes. However, other factors might have an impact on

the complexity of a TFPG, and finding a good metric is an open research question.

9.2 TFPG as an Abstraction of the Plant

The analysis discussed in the previous Section was focused on the TFPG in isolation. However, TFPGs are used to capture the behavior of a plant, and can be considered as an abstract version of the plant. The plant is abstracted in order to focus on one particular aspect of the system: failure propagation time. The abstraction is simpler than the original system. By using a simpler model, we can improve the performance of the reasoning tasks. Therefore, we would like to use TFPG as the base for the FDI design of the plant. In this thesis, we presented several techniques to perform reasoning on top of transition systems. In order to apply the techniques for diagnosis, diagnosability, and diagnoser synthesis on the TFPGs, we need to reduce them to transition systems. This transformation is used within the FAME project (Chapter 10), to perform synthesis from TFPGs.

9.2.1 TFPG as a Transition System

In order to capture the TFPG as a discrete-time transition system, we discretize its timed behavior[2]. The timed behavior of the TFPG is discretized by defining an atomic unit of time, and allowing all behaviors to occur only with timings that are multiple of the basic unit. The unit of time is called *sampling time* (δ). By breaking down possible evolutions of the TFPG according to the sampling time, we obtain a *discretized TFPG* (dTFPG). The dTFPG is a transition system in which each transition takes exactly one sampling time δ. For example, let us consider a sampling time of 0.5,

[2]The final (working) version of the encoding has been developed and implemented by Benjamin Bittner. My contribution focused on the definition of the problem, examples and an initial idea for the encoding.

and an edge with $tmin = tmax = 5$. The same edge will be encoded in the dTFPG as requiring 10 discrete time-steps to propagate.

```
MODULE edge (tmin, tmax, source, target,
             trans_type, time_tick, system_mode_is_compatible)
    VAR counter : 0..tmax;

    -- Failure has propagated to source node but not to the target node (yet)
    -- and the system is in a compatible mode
    DEFINE is_active := source.is_active & !target.is_active & system_mode_is_compatible;

    -- The edge is active and within the propagation time
    DEFINE can_fire := is_active & counter >= tmin & counter < tmax;

    -- The edge is active and has reached tmax!
    DEFINE must_fire := is_active & counter = tmax;

    ASSIGN init(counter) := 0;
    ASSIGN next(counter) := case
        next(!is_active) : 0;                    -- Edge is not active (reset)
        counter < tmax & time_tick : counter + 1;-- Increment
        TRUE : counter;                          -- Keep as-is
        esac;
```

Figure 9.5: Simplified SMV Module for an Edge.

Each edge of the TFPG is translated into a component containing a counter (Figure 9.5). The counter keeps track of how long the edge has been active. Once the counter is within the propagation time, it indicates to the target discrepancy node that it can activate. The counter is reset during mode change . Depending on the type of the discrepancy, we define when it can or must activate, by considering the incoming edges (Figure 9.6). We distinguish between 3 types of transitions: mode-change, activation, and timed. A mode-change transition indicates the change of mode and the corresponding (de-)activation of edges and corresponding counters. An activation transition indicates that a new discrepancy has been activated. This makes it easier to handle 0-delay propagations, and activation of new

edges. Finally, a timed transition (`time_tick` in the figures) is used to make time progress; during this transition counters are increased for all active edges that have not completed the propagation.

```
MODULE node (can_fire, must_fire, time_tick, instant_trans_type)
    VAR is_active: boolean;

    ASSIGN init(is_active) := FALSE;
    ASSIGN next(is_active) := case
        instant_trans_type != NODE_ACTIVATION : is_active;
        must_fire : TRUE;
        can_fire : {FALSE, TRUE};
        TRUE : is_active;
        esac;
```

Figure 9.6: Simplified SMV Module for a Discrepancy.

Failure mode nodes can activate in a non-deterministic way during activation transitions. Figure 9.7 presents a snippet of SMV code encoding a part of the TFPG for the BSS (Figure 9.2). Notice how the semantics of `AND`, and `OR` is encoded in the `can_fire`, `must_fire` definition.

The conversion of a TFPG into a dTFPG makes it possible to apply numerous techniques from the formal verification domain, including diagnosability analysis, verification, and synthesis of diagnoser. The main limitation of this approach is given by the choice of the sampling time. The size of the resulting dTFPG depends on the sampling time and on the constants involved in the edges. Choosing a sampling time that is too small, can lead to a dTFPG that requires thousands of discrete steps to perform a single propagation. On the other hand, a sampling time that is too big, might affect the accuracy of the dTFPG. In fact, multiple traces of the TFPG will be collapsed into a single trace of the dTFPG. This can lead to situations where traces that were diagnosable in the TFPG become non-diagnosable in the dTFPG. A possible solution to these problems would be to allow time-steps to consume more than one time unit, or use a richer

```
MODULE main
...
VAR or_node_S1_WO : node(-- Can Fire?
                         (edge_B1_DEAD_to_S1_WO_EDGE4.can_fire|
                          edge_B2_DEAD_to_S1_WO_EDGE12.can_fire|
                          edge_Sens1_off_to_S1_WO_EDGE13.can_fire),
                         -- Must Fire?
                         (edge_B1_DEAD_to_S1_WO_EDGE4.must_fire|
                          edge_B2_DEAD_to_S1_WO_EDGE12.must_fire|
                          edge_Sens1_off_to_S1_WO_EDGE13.must_fire),
                         -- Other Info
                         time_tick, instant_trans_type);

VAR and_node_System_Dead : node(-- Can Fire?
                                ((edge1.can_fire | edge1.must_fire) &
                                 (edge2.can_fire | edge2.must_fire) &
                                 (edge1.can_fire | edge2.can_fire)),
                                -- Must Fire?
                                (edge1.must_fire &
                                 edge2.must_fire),
                                -- Other Info
                                time_tick, instant_trans_type);
...
```

Figure 9.7: (Partial) SMV encoding of the TFPG of the BSS.

formalism like timed transition systems.

9.2.2 Abstraction of the Plant

Using the TFPG as an abstraction of the plant opens up the question on the quality of the abstraction. Can we use a diagnoser obtained from the abstraction, on top of the real plant? *How good* is the abstraction of the plant? In particular, the TFPG needs to capture all the *interesting* behaviors of the plant. To achieve this, we introduce the general idea of *cross diagnosability*.

Cross Diagnosability makes it possible to relate the observable traces of two systems, and thus leads to *diagnoser reuse*, i.e., the possibility of

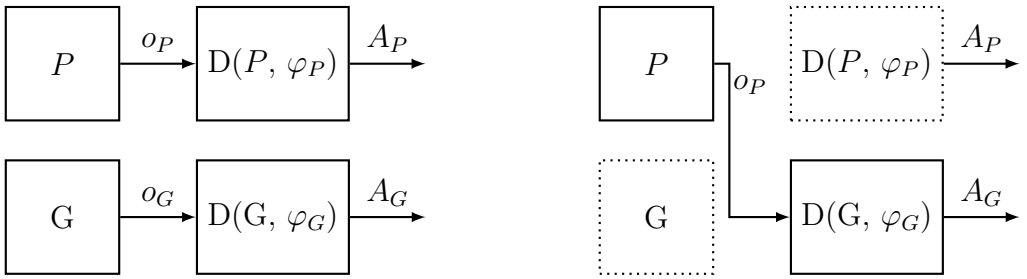

Figure 9.8: Systems and Diagnosers connections

using a diagnoser that was designed for one system and apply it on another system. In our case, we design a diagnoser for a dTFPG and use it on the plant. However, the second system could also be obtained by constructive abstraction techniques [91]. Figure 9.8 schematizes the idea of diagnoser reuse. A plant P and a dTFPG G are given, together with two alarm specification φ_P and φ_G. P and G are not necessarily related, and the two specifications can be arbitrary. We can thus build the diagnosers $D(P, \varphi_P)$ and $D(G, \varphi_G)$ (Figure 9.8 – Left). Let us assume, for now, that P and G have the same observable language. Without additional assumptions, we want to know whether we can replace $D(P, \varphi_P)$ with $D(G, \varphi_G)$ and still obtain the same diagnosis (Figure 9.8 – Right). More formally, whether (up-to renaming) $D(G, \varphi_G)$ is a correct and maximal diagnoser for φ_P in P. In particular, by considering the ASL_K formulation of the alarm condition φ_P (Section 6.3), we ask whether:

$$P \times D(G, \varphi_G) \models \varphi(A_P, \varphi_P)[A_P/A_G]$$

Requiring that both system have the same observable language, allows us to guarantee (by construction) the *compatibility* of the diagnosers. This means, that the diagnosers are able to accept in input the same set of observable traces. Since we are interested in a particular direction (using a diagnoser of G on P), we can weaken this requirement and require that every trace of P can be mapped into a trace of G. For simplicity, we

now assume that the mapping is an identity function, and thus that the observable language of the plant is a subset of the observable language of the TFPG ($\mathcal{L}_O(P) \subseteq \mathcal{L}_O(G)$). In this way, every trace of the plant has a corresponding trace in the TFPG.

Diagnoser reuse opens the possibility of applying the following process:

1. Design G and φ_G
2. Synthesize $D(G, \varphi_G)$
3. Check/ensure that:
   (a) $P \times D(G, \varphi_G) \models \varphi(A_P, \varphi_P)[A_P/A_G]$ and
   (b) $D(G, \varphi_G)$ is compatible with P.

Since the diagnoser construction (Step 2) is an expensive step (Chapter 8), we want to check whether $D(G, \varphi_G)$ is going to be a diagnoser for P before constructing it, therefore, we extend the concept of diagnosability across the two systems.

When testing diagnosability via the twin-plant (Chapter 7) we consider pairs of traces from the *same* system. The intuition behind cross diagnosability is to consider pair of traces that come from two different systems (e.g., the plant and the TFPG). As an example, the definition of bounded delay diagnosability (Section 5.4) is extended as follows:

Definition 29. *(Bounded Delay Cross Diagnosability) Given two plants P and P_G, two diagnosis condition β and β_G, a recall $\mathcal{R}$, two sets of observables E_O and E_{O_G} (respectively of P and P_G), and a mapping function $\gamma : E_O \to E_{O_G}$, we say that* BoundDel(A, β, d) *is* cross system diagnosable *w.r.t P_G and the diagnosis condition β_G in P iff forall $(\sigma_P, i) \in P$ s.t. $\sigma_P, i \models \beta$ there exists k s.t. $i \leq k \leq i + d$, $ObsPoint(\sigma_P, k)$ and for all $(\sigma_G, l) \in P_G$, if $ObsPoint(\sigma_G, l)$ and $obs^{\mathcal{R}}_{E_{O_G}}(\gamma(obs(\sigma^k_P))) = obs^{\mathcal{R}}_{E_{O_G}}(obs(\sigma^l_G))$, then there exists j s.t. $l - d \leq j \leq l$, and $\sigma_G, j \models \beta_G$.*

This definition extends Definition 14, by embedding the mapping γ in the definition of *ObsEq*. Intuitively, we want to compare every trace of the plant in which β occurred, with every trace of P_G, that has the same observations up-to mapping.

To keep the definition simple, we require the alarm specification to be of the same type on both plants. The above definition can be extended to the other alarm conditions, and be generalized to trace diagnosability.

To test the system diagnosability, we can extend the coupled twin-plant approach. Instead of using two copies of the same plant, we use one copy of the plant and one of the TFPG, and then verify the properties as described in Chapter 7. In practice, this provides us with an effective approach to verify whether a dTFPG captures all behaviors of interest of the plant that it is modeling.

The process is not limited to TFPGs, and it can be applied on a generic system G. In particular, G can be an abstraction of P, obtained via techniques such as predicate abstraction [109]. The concept of cross diagnosability generalizes the approach described in [91], in which sufficient conditions are given for abstractions techniques that guarantee cross diagnosability (called abstract diagnosability).

9.3 Chapter Summary

We presented a novel characterization of TFPGs based on symbolic techniques. We explored several important reasoning tasks that aim at increasing the confidence of the designer in the TFPG model, thus guaranteeing that the online reasoning tasks (e.g., diagnosis) can be effective.

Our framework provides a way to describe and perform model validation, refinement testing, diagnosis and diagnosability in a unified way. Other reasoning tasks can be defined in the future by relying on the SMT

solvers to perform the reasoning, without implementing ad-hoc reasoning algorithms. Finally, we experimentally show that these techniques are applicable on TFPGs of considerable size.

We describe an approach to convert a TFPG into a transition system (dTFPG), and then clarify the relation between the TFPG and the plant. In order to exploit the dTFPG and certify its effectiveness, we propose the concept of diagnoser reuse and cross diagnosability.

An interesting open point concerns the possibility of automatically generating a TFPG from a given system model, and [27] represents a first attempt in this direction.

Chapter 10

Industrial Experience

This thesis has been motivated by the industrial need to improve the design process of FDIR systems. The FDIR design is currently post-poned until late phases of the system design. This leads to designs that are sub-optimal, and mostly based on past successful strategies. Validation of such solutions is a difficult task and is currently mostly performed manually. Moreover, late changes to the plant design might have drastic impact on the FDIR strategy. Similarly, in order to achieve the FDIR objectives, it might be necessary to modify the system design. The impact of these changes might be profound, and potentially delay launch.

The European Space Agency (ESA) started several projects to try and improve the FDIR development and verification process. The goal is to increase the confidence on the FDIR design, simplify its certification, and shorten the design loop. In this Chapter, we report on two projects in which we were involved: AUTOGEF and FAME.

We first provide some context to these projects, giving an overview of the COMPASS project, and highlighting the starting point for both AUTOGEF and FAME (Section 10.1). Section 10.2 discusses a general process for the use of formal techniques in the development of FDIR. In Section 10.3, we show how this process has been applied, by our industrial

partners, on a case-study based on ExoMars in both AUTOGEF and FAME, and discuss the technical differences in both projects.

The contribution of this Chapter is to present the AUTOGEF and FAME projects. These projects served as starting point for most of the work in this thesis. Additionally, we discuss the challenges involved in FDIR design, and describe a possible process to introduce formal techniques in the design flow. This Chapter extends the material presented in [25].

10.1 COMPASS

The design of critical systems in aerospace is a complex and highly challenging task, as it requires assembling heterogeneous components, implemented either in hardware or in software, and taking into account their interactions.

The ESA project COMPASS tried to address some of these problems by introducing a tool-set [35, 61] to apply formal verification to system-software co-engineering. The tool-set supports model-based development and verification of aerospace systems. It takes in input a formal model of the system, and can perform several types of analysis: requirements analysis, functional verification, safety assessment, performability evaluation, and diagnosability. The modeling language is called System-Level Integrated Modeling (SLIM) language [39] and it is based on the Architecture Analysis and Design Language (AADL) [80] and its Error Model Annex [81]. AADL has become a standard in the industry for the description of systems. The fact that SLIM is derived from AADL should simplify the adoption of the technologies provided by COMPASS within the industry. The SLIM language can be used to model discrete, timed, hybrid and probabilistic behaviors. The architectural language makes it possible

to break down complex system into a hierarchy of components, that can be independently modeled and validated.

The main focus of COMPASS was on the design of the plant. Support for FDIR design was mostly limited to techniques for verification (e.g., model-checking) of FDIR components. In order to properly design FDIR components, we need to be able to formally specify the requirements and expectations that we have on the FDIR. Unfortunately, there is no defined FDIR development process for aerospace that coherently addresses the full FDIR life-cycle, including the corresponding verification and validation perspective. An effective FDIR development strategy needs to start in the early system development phases, and take into account the design and RAMS (Reliability, Availability, Maintainability, Safety) data from both software and system perspective. However, the results of Software and Hardware RAMS activities (e.g., FTA and FMEA), become available late in the process, leading to late initiation of the FDIR development, which has a detrimental effect on the FDIR maturity. Finally, there is a conflict between the bottom-up and top-down approaches: FMEA (bottom-up) can not be completed until system design has sufficient levels of details, whereas FTA (top-down) does not guarantee that every possible component failure mode which contributes to system failure has been considered. The AUTOGEF and FAME projects were launched in order to try to fill this gap in the process and technologies. The AUTOGEF [7] project was a collaboration between ESA, GMV, Thales France and FBK. The project aimed to demonstrate that synthesis approaches can allow for effective automated FDIR development in accordance with the dependability requirements. The main focus was on the model-based automated FDIR model generation. The FAME [79] project was a collaboration between ESA, Thales Italy, Thales France and FBK. The goal was to take the informal flow introduced in AUTOGEF, consolidate it and cast it within

industrial practices and standards. At the same time, the modeling formalism went from discrete events to timed behavior, thus highlighting the importance of propagation times when considering faults, and the need for representation formalism such as TFPGs.

Both projects provide an extended version of the COMPASS tool-set, enabling the application of the techniques discussed here on top of existing SLIM models. Overall, the three projects (COMPASS, AUTOGEF and FAME) span across more than 5 years of research.

10.2 FDIR Development Process

The design of the FDIR depends on the availability of detailed information on the system: which faults should be considered, how they can impact and propagate to other components, and which options are available for recovery. For this reason the FDIR design is usually postponed until the system design has been frozen, thus leaving little time to perform FDIR design and validation. To address these shortcomings, we propose a novel process for FDIR design. This process aims at enabling a consistent and timely FDIR conception, development, verification and validation. It enables the specification and analysis of failure propagation using fault propagation models, the possibility to specify a set of relevant requirements for FDIR, and to model, or synthesize, FDIR components that comply with the requirements. Finally, it enables verification of the effectiveness of the FDIR.

Integrating a formal modeling tool (such as COMPASS) with an automated way to synthesize the FDIR can improve the FDIR design process. Automated synthesis techniques generate solutions that are correct by construction. Therefore, it becomes possible to perform what-if analysis by studying the effect of different FDIR requirements on the overall design.

Moreover, the synthesized FDIR can be used as placeholder, until a more detailed design is available.

The process is composed of the following phases:

Analyze User Requirements The requirements at the system level are captured by developing a formal model and associated properties. Moreover, relevant mission phases and operational modes are identified.

Perform Timed Failure Propagation Analysis Timed impact of faults is studied. A Timed Failure Propagation Model (e.g., a TFPG) is developed using fault trees and FMEA tables, and the associated system model.

Define FDIR Objectives and Strategies The specification of the FDIR is defined by choosing which conditions to monitor, how to react, severities, etc.

Design FDIR The FDIR is designed in order to satisfy the FDIR requirements.

Table 10.1 shows the steps of the process, and how they can be mapped to the tool support provided by COMPASS and extensions.

Formal design of FDIR can be carried out in parallel (and not as an alternative) to the classical process. This makes it possible to introduce the process within existing industrial processes. FAME and AUTOGEF provide an implementation defined on top of COMPASS, thus inheriting the modeling language SLIM. However, we believe that the process is abstract enough as to be implemented on top of other technologies.

Table 10.1: Process break-down

Phase	**Steps**	**COMPASS**
Analyze User Requirements	System Modeling & Fault Extension	Formal system modeling – nominal and faulty behavior (in SLIM); automatic model extension
	Formal Analyses	Derive requirements on FDIR design
	Mission Modeling	Definition of mission, phases, and spacecraft configurations
Perform Timed Failure Propagation Analysis	Formal Analyses	Derive information on causality and fault propagation (input for TFPG modeling)
	TFPG Modeling/Synthesis	TFPG modeling, editing, synthesis
	TFPG Analyses	TFPG behavioral validation, TFPG effectiveness validation
Define FDIR Objectives and Strategies	FDIR Requirements Specification	Modeling of FDIR objectives and strategies, definition of pre-existing components to be re-used, and FDIR hierarchy
Design the FDIR	FDIR Modeling/Synthesis	Formal modeling and automatic synthesis of FDIR
	Formal Analyses	FDIR effectiveness verification

10.2.1 Analyze User Requirements

The *formal model of the system* is developed using the SLIM language. SLIM is an architectural language, that can capture a wide range of dynamics: discrete, continuous, hybrid, and probabilistic.

The nominal model can be extended in order to introduce faults, using *error models* and fault injection. During this process, the user only needs to specify the nominal model behavior, the error model of a component, and select which components can be affected by the fault. The tool takes care of automatically extending the model introducing the faults. The user specifies the *observability* of the system, i.e., which event and data ports can be observed and used by the diagnoser. For recovery, instead, sets of

events that can be used to guide the system are marked as *recovery actions*. These events will be used by the recovery synthesis algorithms to try to identify a recovery plan.

The AUTOGEF project focuses on models that are finite and discrete-time. Both events and (finite) data can be used as observables, and the diagnoser is assumed to have perfect recall. The synthesis techniques described in this thesis were motivated by AUTOGEF, and thus perfectly fit within this setting. The FAME project, instead, uses models that are continuous-time.

By developing the model within the COMPASS tool-set, it is possible to validate the model before proceeding with the FDIR specification and design. The system designer can apply model-checking techniques to validate the system behavior, and perform RAMS (Reliability, Availability, Maintainability, Safety) analysis, thus validating the model of the system. By using Fault Trees and FMEA tables, the system designer can identify critical faults, that need to be handled by the FDIR.

The FDIR needs to behave differently in different moments of the mission. Depending on the situation, faults severity might change, and some recovery actions might not be allowed. This calls for an FDIR specification that is aware of the current *mission phase* and *operational mode*. In order to capture this information, we ask the system designer to define a *Mission Specification*. The mission specification contains the description of the relevant mission phases and the associated operational modes. Moreover, it associates possible operational modes to one or more mission phases. In order for the FDIR to use this information, it needs to be able to understand (at run-time) what the current phase and mode is. The system designer needs to specify some observable condition (i.e., a condition expressed over observable variables only) that identifies the current phase and mode.

Finally, we require that the user also defines the *spacecraft configura-*

tions. Each spacecraft configuration can be associated with one or more operational modes. The diagnosis and recovery engines will assume that, whenever the system is in the given phase and mode, the spacecraft is within one of the given configurations.

10.2.2 Perform Timed Failure Propagation Analysis

Starting from safety artifacts such as fault trees and FMEA tables, the designer is able to study how a fault can impact the system. Using model-checking techniques, it is possible to verify the minimum and maximum time required by a failure to propagate within the system. This information can be captured into a TFPG (Chapter 9). TFPGs model how failures propagate, affecting various monitored and unmonitored properties, and are thus an abstract view of the underlying system.

In FAME, we support the loading, syntactic verification, displaying, and editing of TFPGs. Nodes are defined using basic expressions over system variables. This makes it possible to map executions of the system to traces of the TFPG. FAME allows checking whether the system exhibits failure propagations that are not captured by the TFPG (*behavioral validation*). If wrong values are present, a counter-example is produced to guide the user in the refinement process. If no counter-example is found, the analysis guarantees that the timing values are correctly specified. The tool also allows checking the TFPG adequacy as a model for diagnosis, using diagnosability analysis (called *(diagnosability) effectiveness validation*). This analysis enables the identification of the failure modes that are not diagnosable. Finally, a technique for automatic generation of the structure of the TFPG is available. This technique exploits the fault tree computation engine to extract discrepancies relations, starting from the system model and discrepancies definition.

On the other hand, in AUTOGEF, we are not concerned with the

timed behavior of the faults, and therefore this phase is not considered.

10.2.3 Define FDIR Objectives and Strategies

The definition of the FDIR Specification is driven by the mission phases and modes, the spacecraft configurations, and the faults. The FDIR Specification is divided into two main areas: Fault Detection (FD) and Fault Recovery (FR).

In the FD specification, it is possible to consider a subset of all the faults that can occur in the system. Not selecting a fault might be justified by the limited impact of the fault on the system, or simply by the desire of starting with a simpler design in order to perform multiple iterations and gradually increase the number of faults considered. Each selected fault is associated with an alarm. The definition of the alarm can be either delegated to the synthesis engine, or mapped to an existing alarm of the system model.

The choice of which faults to consider should be driven by the results of the RAMS analysis, and the potential impact of each faults towards the mission objectives. The diagnosis conditions are assigned to the target faults. Using the results of the failure propagation analysis, it becomes possible to decide the acceptable delay for each alarm condition.

The FR specification is composed of a table associating each triple of Alarm, Phase and Mode to a recovery goal. This provides several independent recovery problems. Each of these problems can be enriched by adding information concerning the severity of the alarm (within the given phase and mode), the constraints on the target spacecraft configuration to reach, and limits to the set of actions that can be used.

Similarly as done for the FD, the definition of the recovery procedure can be delegated to the synthesis engine, or it can be given by identifying an existing component (within the system model) that can carry out the

recovery.

Several information need to be provided within the FDIR specification. *Patterns* are used to simplify the process. Patterns can be used to pre-fill the FD and FR table. For example, default recovery actions can be associated with a given alarm.

10.2.4 Design the FDIR

The FDIR can be designed starting from the FDIR Specification. The FDIR needs to cover all the specified requirements for each mission phase/operational mode. Moreover, existing components that were included in the FDIR Specification for some phase/mode, need to be combined with the final FDIR design.

Once the design is completed, it can be verified by applying model checking techniques. In particular, we can verify that faults are detected correctly and recoveries lead the system to the desired state.

In both AUTOGEF and FAME, both the FD and FR components are automatically synthesized, starting from the available information. The mission specification and the link between alarms and recoveries are captured by creating the *FDIRConf* component (Figure 4.4). Having this intermediate component simplifies the integration of generated and predefined FDIR components.

Since AUTOGEF assumes a discrete-time system, while FAME assumes a continuous-time one, we approach the synthesis in two different ways. Nevertheless, the outcome is an FDIR that satisfies the FDIR requirements by construction. The synthesis engines have been developed within the xSAP tool [158].

AUTOGEF and Discrete-Time FDIR The FD synthesis follows the belief explorer construction explained in Chapter 8 for asynchronous systems.

For each alarm in the FD specification, we create a FiniteDel alarm specification, and require a trace diagnosable and maximal alarm. Therefore, the output of the synthesis process is a three valued alarm (Section 8.1), indicating that i) the fault *did* occur, ii) the fault *did not* occur iii) it is impossible (with the given observations) to know *whether* the fault did or did not occur.

The FR synthesis uses techniques coming from planning under partial observability [19]. The outcome of the synthesis is an automaton that, when activated, executes a plan with branching points. In those branching points, the FR uses observations coming from the system to decide how to proceed. If a plan is found by the synthesis process, it is guaranteed to always be able the take the system to the target mode and configuration.

FAME and Continuous-Time FDIR To deal with the continuous-time nature of the system models used in FAME, we use an abstraction technique. For the synthesis of the FD, we use the TFPG as an abstract view of the system. This means that we abstract the FDIR requirements, in order to only talk about *failure mode* appearing in the TFPG, and *observable discrepancies*. By providing a *sampling rate* to the engine, we are able to discretize the TFPG and obtain a discrete-time model (as described in Chapter 9.2). The dTFPG is a finite state system, therefore we can apply the synthesis algorithm and obtain a discrete-time diagnoser. The discrete-time diagnoser is then converted into a timed automata, that is controlled by the chosen sampling rate. Additionally, we introduce a component that is tasked with translating traces from the system into traces of the TFPG (`Sys2TFPG` in Figure 10.1).

The FR component is built by using techniques of conformant planning [144]. The reasoning is performed on an abstract version of the system [109], obtained by using predicate abstraction techniques. A recovery

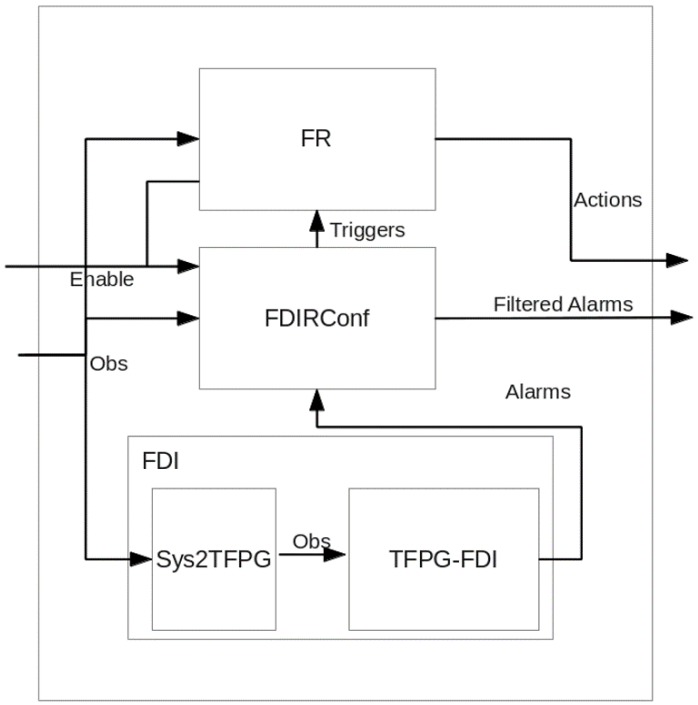

Figure 10.1: FAME FDIR Architecture

plan consists of a sequence of actions that are guaranteed to bring the system to the target condition. A conformant plan does not have branching points. However, since we are working on timed systems, the recovery plan can include operations for waiting for a certain amount of time. After synthesis, the FD and FR components are connected as shown in Figure 10.1.

10.3 Case Study: ExoMars

The evaluation for both AUTOGEF and FAME was performed by Thales Alenia Space on a sub-set of the Trace Gas Orbiter (TGO) of the ExoMars project [62]. The ESA ExoMars system will be launched in 2016 and will arrive to Mars approximately 9 months later. The system is composed of a spacecraft that will carry an Entry and Descent Module (EDM) demon-

strator. During the transit from Earth to Mars, the TGO will carry and provide power and other services to the EDM. The release of the EDM will take place prior to the critical Mars Orbit Insertion (MOI) maneuver by the TGO. After capture by Mars gravitation field, the TGO will orbit around Mars and provide support to the EDM. Once the EDM surface operations are completed, the TGO will start a science data acquisition phase. Near the end of this phase, the 2018 mission should arrive to Mars, relying on the TGO for support.

The case study was chosen since it provides an opportunity for evaluating all the aspects of the approach. In particular, it presents a complexity level that is representative of the classical complexity level in this domain.

The ExoMars mission can be divided into several mission phases, but in our case study we only consider the Mars Orbit Insertion (MOI). In this phase, several operational modes are used: nominal, safe, and degraded. Degraded modes are used by the FDIR when reconfiguration is required. The main functional chain considered during the case study is the Guidance, Navigation and Control (GNC) function which encompasses sensors, control software, and actuators. The goal of this sub-system is to maintain the correct spacecraft attitude. The faults that were considered are those related to the units that can lead to the main feared event considered in this study: the loss of spacecraft attitude.

The system architectural and behavioral information, and information concerning the mission phases and operational modes, were used as inputs to model the nominal system.

Feared Event Analysis and FMECA The first activity for safety analysis is the Feared Event Analysis. We are interested in the feared events coming from the units realizing the acquisition of the spacecraft attitude: the Inertial Management Units (IMU). Three possible failure modes are considered:

Item	Sensor Fault	Local Effect	System Effect
IMU_001	Signal is too low	Continuous self-reset of IMU	No measure is sent
IMU_002	Output is biased	None	Biased output from sensor channel
IMU_003	Output is wrong	Loss of RLG dither control	Wrong output from sensor channel

Figure 10.2: FMECA Table

No measure, *Biased measure* and *Wrong measures*. The only failure mode that is not diagnosable is biased measure, while absence of measure and wrong measures are detectable either by rate control, or by cross-checking with other readings. Using documentation and FMECA from the IMU equipment supplier, the IMU FMECA Items are analyzed and those having impact on the system are selected. Figure 10.2 gives a selection of three FMECA items of the IMU equipment. The local and system effects can be matched with the failure modes identified in the previous feared event analysis.

Failure Propagation Modeling To simplify the analysis, subsets of system failures have been considered, e.g., in cold redundancy the failures of nominal equipment are analyzed independently of the failures of the redundant equipment. Let us consider the IMU equipment. In any mission phase where this unit is used, the failures propagate into the system, since no FDIR prevents the propagation (yet), and has an impact on the spacecraft attitude, leading to the feared event that we are interested in: loss of the spacecraft attitude. This fault propagation is modeled in the SLIM model by implementing for each function the effect on its outputs given wrong input values. Since fault propagation may not be immediate, it is important to consider timing information. Function implementations therefore introduce delays in this propagation.

Specification of FDIR A set of requirements coming from the ExoMars TGO project were analyzed to produce FDIR objectives, strategies, and specifications. The complete FDIR specification for the case study in FAME defines one alarm for each of the selected faults (Figure 10.2) and for each of the IMUs, thus providing a total of 6 alarms in the specification. Each alarm is associated with a recovery requirement for each possible phase and mode combination. In total, this results in 24 recovery requirements, 12 for each IMU unit: the nominal (IMU1) and the redundant one (IMU2). In AUTOGEF, a few additional components were considered, leading to 10 alarm and 27 recovery requirements.

Timed Fault Propagation Graph Modeling Instead of building a global TFPG that would cover all failures of the system for all modes, we build several TFPGs, covering the failures of respectively the nominal and the redundant IMU. The TFPG model for the nominal IMU is defined starting from the SLIM model enhanced with timing aspects and the error model (Figure 10.3). On this TFPG we can see the three failure modes of the nominal IMU equipment propagating in the system. Some of the discrepancies are observable, whereas some are not. Associations between discrepancies modes and system model define the relation between nodes in the TFPG and the original system. The TFPG was validated using behavioral and effectiveness validation analysis provided by the tool, thus verifying that the TFPG properly capture diagnosability of the system.

FDIR Synthesis The FD synthesis resulted an FD SLIM module that encodes a finite state machine with 2413 states. The FR synthesis resulted in an FR SLIM module with recoveries (6 recoveries out of 9 are found). The missing recoveries identify situations in which there is no strategy that can guarantee the recovery. Although it might be possible to find strategies

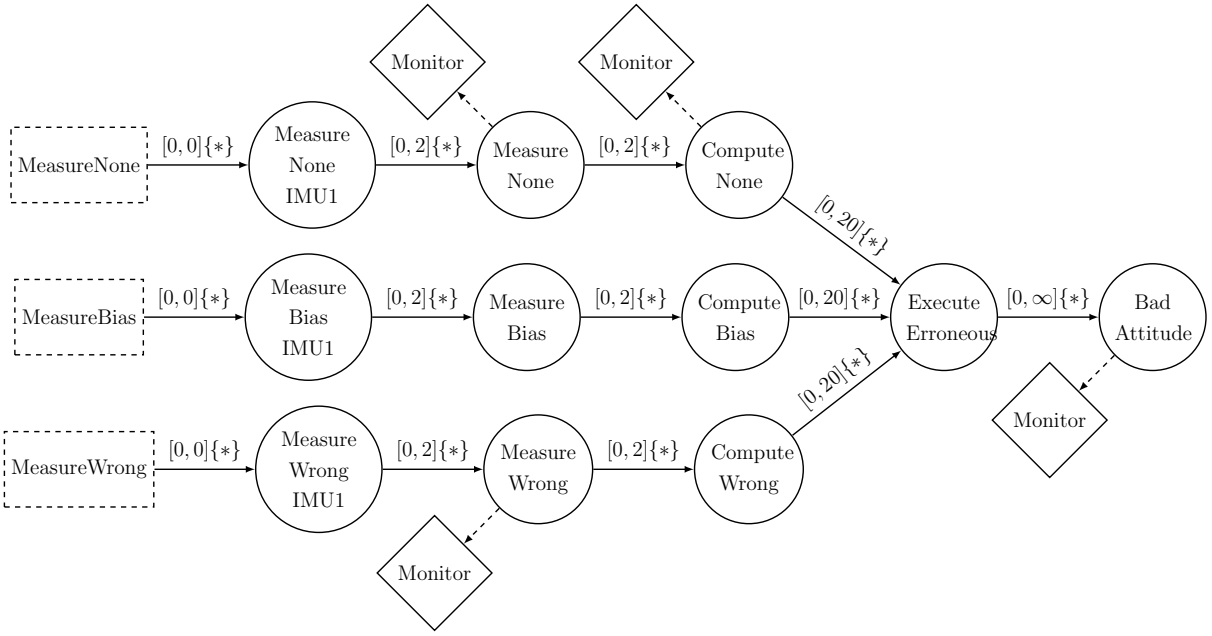

Figure 10.3: Case study TFPG (Nominal IMU)

that would work under certain circumstances, the tool focuses on finding solutions that always work.

In AUTOGEF, the scope of the modeling was slightly broader and included also other components of the system (namely the On-Board Controller, and the Thrusters). All 10 alarms for the FD were correctly synthesized, leading to a diagnoser with more than 700 states. The synthesis of the Fault Recovery module managed to identify 18 (out of 27) plans. The FAME FD has more states than the one from AUTOGEF. The reason is the size of the underlying dTFPG model. Although, the TFPG is an abstraction of the plant, when discretizing it to obtain the dTFPG, we end up with a system with many states. In turn, this leads to a bigger diagnoser. This does not go against the idea of performing the abstraction in the first place. In fact, without abstraction, we currently would not be able synthesize any diagnoser.

In both cases, the resulting FDIR and extended system model, were further analyzed using model-checking techniques provided by the COMPASS

tool-set, and the effectiveness of the solution was verified.

10.4 Chapter Summary

In this Chapter we presented two ESA-funded projects that strongly influenced this thesis. The framework for specification of FDIR requirements is at the base of ASL_K (Chapter 6), AUTOGEF fostered the need for the FDI synthesis techniques (Chapter 8) and FAME pushed us to develop validation techniques for TFPGs (Chapter 9). Moreover, these projects motivated the FDIR design process presented in this Chapter. We also discussed the application of these techniques on a case-study. The process and tooling have been successfully evaluated by the industrial partners within the ESA projects AUTOGEF and FAME. The tools supporting those projects, although research tools, received positive feedback from the industrial partners. In particular, they helped to formally define and analyze different options for system and FDIR design.

There are several interesting technical directions for future work. The TFPG modeling process is still mostly manual, and should be supported by TFPG synthesis algorithm. An initial approach to compute the structure of the TFPG was integrated in FAME. This provides some support to the designer, but more work (e.g., [27]) is needed to avoid relying on manual inputs. From the point of view of the process, it would be important to trace requirements through the different phases of the formal development. Friendliness towards traceability of the requirements might be an important obstacle to overcome before these tools are applied in industry. In the same direction, it would be interesting to integrate the COMPASS tool-set with other modeling tools used in industry (e.g., Melody Advance, used in Thales). Since modeling is a time- and cost-consuming activity deriving the SLIM model from other formalism could be a first step to wider adoption

of these techniques.

Chapter 11

Temporal Epistemic Logic Model-Checking

Temporal Epistemic Logic (TEL) is a modal logic combining operators for the evolution of time, and the representation of knowledge. In this thesis, we focused on the application of TEL to the diagnosis domain. Nevertheless, other interesting applications of TEL exist, for example in the domain of information security [9], or cryptographic protocols [30], where we are interested in guaranteeing that some information will remain private, even if some public information is shared.

There are multiple variants of TEL, depending on the type of temporal and epistemic logic being considered. From the temporal side, we distinguish between linear-time and branching-time. The epistemic operator, instead can be characterized by several types of recall. In this thesis we focus on LTL, and consider multiple types of recall, thus obtaining the KL logic [93]. In particular, we focus on the fragment KL_1 of the logic, in which we do not allow nesting of the epistemic operator. This logic is widely used in the literature [78, 154, 108, 14, 107, 14].

Concerning the properties of the models being studies, existing tools deal with finite state systems. To properly capture the target domains of physical systems, we need to be able to use infinite state models. There is

currently a lack of tools for temporal epistemic model-checking in infinite state systems, and works in this area are mostly limited to theoretical ones. At the same time, infinite state model-checking for transition systems has become a consolidated area of research in the formal verification community [67, 16, 47], where efficient SAT/SMT [12] based algorithms have been developed. Although, in the general case, this is an undecidable problem, these algorithms (e.g., IC3 [51]) are able, in practice, to deal with infinite state models. Moreover, when these techniques are applied to finite state models, they usually can handle larger models than the algorithms based on Binary Decision Diagrams (BDDs [46]).

Our goal is to exploit existing verification algorithms (e.g., IC3) to model-check KL_1 over finite/infinite state synchronous transition systems under observational semantics. To achieve this, we propose two approaches that work by reducing KL_1 to LTL. First, we consider an *eager* approach, that works by performing an up-front computation of the states that satisfy the epistemic subformula. This approach relies on parameter synthesis techniques, and works, in practice, for finite and infinite state systems. Interestingly, this approach is sensitive to the number of observable variables. Therefore, if the model consists of few observable variables, we manage to obtain reasonable performances. Unfortunately, the approach does not scale on models with a significant number of observables. Our second solution is characterized by a *lazy* approach. The computation of the states satisfying the epistemic subformulae is not carried out up-front. Rather, we rely on a counter-example guided abstraction refinement (CEGAR) of the property. Similarly to the lazy approach in Satisfiability Modulo Theories, the epistemic atoms are initially treated as Boolean variables, and incrementally axiomatized as a result of proving the spuriousness of the counterexamples.

In Section 11.1 we introduce the fragment $InvKL_1$, and provide exam-

ples of its usage in the literature. The Eager approach is presented in Section 11.2.

Section 11.3, introduces the Lazy approach, several optimizations, and the generalization to KL_n. Section 11.4 provides a detailed experimental analysis.

The contributions of this Chapter are:

1. Identify the fragments KL_1 and $InvKL_1$, and show their relevance in the literature and in practice;
2. Propose and implement the first practical approaches (Lazy and Eager) for model-checking KL_1 on infinite-state transition systems;
3. Introduce several optimization for the Lazy approach, that are fundamental to achieve competitive performances;
4. Provide a detail comparison of both approaches; additionally, we compare the approaches against existing tools for finite-state KL_1 model-checking.

The material of this chapter is based on [49]. With respect to [49], we show how to extend the Lazy approach to KL_n, and we substantially extended the explanation of the Eager approach and its experimental evaluation.

11.1 KL_1 and $InvKL_1$

KL_1 is the restriction of KL (see Section 2.4) in which there is no nesting of the epistemic operator. *Epistemic invariants* of KL_1 ($InvKL_1$) are the formulas that fall into the following syntactic fragment (ϕ):

$$\phi := G\psi \ , \quad \psi := p \mid \psi \wedge \psi \mid \neg\psi \mid K_A\gamma$$

$$\gamma := p \mid \gamma \wedge \gamma \mid \neg\gamma \mid X\gamma \mid Y\gamma \mid \gamma U \gamma \mid \gamma S \gamma$$

Notice that, apart from the top-level G, all other temporal operators can occur only within the K (i.e., γ). All ASL_K specifications (Chapter 6 – Table 6.7) can be expressed in KL_1. In particular, Maximality falls into the $InvKL_1$ fragment. $InvKL_1$ and LTL are sufficient to verify correctness and maximality properties of ASL_K specifications; only diagnosability (and thus trace completeness) requires full KL_1.

Outside of the FDI design domain, $InvKL_1$ (and thus KL_1) is a widely used fragment, despite its simplicity. The following are just a few examples of properties in the literature that are in $InvKL_1$:

- Muddy Children [78]:

$$G(((K_i muddy_i) \vee (K_i \neg muddy_i)) \rightarrow says_i)$$

- Dining Cryptographers [154]:

$$G\big([(K_1 \neg paid_1) \wedge (K_1 \neg paid_2) \wedge (K_1 \neg paid_3)] \vee$$
$$[K_1(paid_1 \vee paid_2 \vee paid_3) \wedge \neg(K_1 paid_2) \wedge \neg(K_1 paid_3)]\big)$$

- Card Games [108]: $G(allred \rightarrow K_{Player1} F(win_1))$

and more examples include the Faulty Train Gate Controller [14], the Gossip Protocol [14], and goals in planning problems [107].

In the context of diagnosis, we did not find situations in which nesting of epistemic operators are needed. In this thesis, we considered a centralized, monolithic FDI. The approach can be extended to encompass a distributed FDI by considering multiple agents and their *distributed knowledge*. A proper formalization of distributed FDI is left as future work, however, we would like to point out that our reasoning engine can handle the distributed knowledge operator D_G. D_G defines the collective knowledge of a group G of agents. Assuming that all agents in G have the same recall, distributed knowledge can be captured by introducing a new agent that has access to all the observations of the group G, thus fitting within the KL_1 fragment.

Voters Example

The type of systems that we consider have a finite number of variables which, however, can have infinite domains. For example, we can have integer or rational values, and use the theory of arithmetic [73] to define the transition relation, as demonstrated by the following *voters* example.

A group of people are called to express a vote. The vote is represented by an unbounded integer value greater or equal than 0. The vote is secret, but the jury can access the sum of all votes (observable). This simple model can be capture by the following transition system:

$$\begin{aligned} Vars &: \{guess \in \mathbb{N}, vote_i \in \mathbb{N}, result \in \mathbb{N}, voted \in \mathbb{B}\} \\ Init &: \neg voted \\ Trans &: (result' = vote_0 + \cdots + vote_n) \wedge \\ &\quad voted' \wedge guess' = guess \wedge vote_i' = vote_i \end{aligned}$$

Can the jury know what somebody voted? We formalize this by checking whether the jury can know what the first voter voted, i.e.,

$$M \models G(voted \rightarrow \neg K_{\{jury\}} vote_1 = guess)$$

where $guess$ is an integer variable and $jury = \{result, guess\}$. Intuitively, for every possible guess, it is not possible for the jury to know that the first voter's vote matches the guess. The counter-example to this specification is a corner case: if everybody votes 0, the jury knows what everybody voted. To prove the property, we need to show that for any value of $guess$, $vote_1$ and $result$ the property holds. Since those variables have an infinite domain, we cannot simply enumerate all solutions, but need to reason symbolically on the set of states.

11.2 Eager Approach

Thanks to the observational semantics, the satisfaction of the formula $K_A\varphi$ in (σ, n) depends only on the state $\sigma[n]$. In particular, if $(P, \sigma, n) \models K_A\varphi$, then $(P, \sigma', m) \models K_A\varphi$ for every trace σ' of P such that $obs_A(\sigma[n]) = obs_A(\sigma'[m])$. We define the *denotation* of $K_A\varphi$ in P (written $[\![K_A\varphi]\!]_P$):

$$[\![K_A\varphi]\!]_P = \{s \in Reach \mid \forall\sigma, \forall n.\ obs_A(\sigma[n]) = obs_A(s) \Rightarrow (P, \sigma, n) \models K_A\varphi\}$$

For every σ, for every $n \geq 0$, $(P, \sigma, n) \models K_A\varphi$ iff $\sigma[n] \in [\![K_A\varphi]\!]_P$. This key observation, allows us to define the satisfaction of epistemic atoms of KL_1 on states, instead that on traces (as done in the semantics definition). A simple way of solving the model-checking problem for KL_1, consists in computing the denotation of the epistemic atoms up-front, and replace the epistemic atoms with their denotation, thus obtaining a pure LTL formula that is equivalent w.r.t. a target partially observable transition system. This allows us to use any existing LTL model-checking algorithm for finite or infinite state systems, such as bounded model checking (BMC) [22] using Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers, or more recent techniques such as IC3 [42].

Recall that a partially observable transition system (POTS) is a transition system, in which we identify some state variables as observables (i.e., $V_O \subseteq V$), and write $P = \langle V, V_O, I, T\rangle$. Let φ be a KL_1 formula containing $K\varphi$ as a subformula. We use $[\![K\varphi]\!]$ to indicate both the denotation of $K\varphi$ and its symbolic characterization expressed over the state variables X of the POTS P.

Lemma 3. *For a POTS P and a KL_1 formula ψ, $P \models \psi$ iff $P \models \psi[K\varphi/\,[\![K\varphi]\!]]$.*

Proof. By definition of denotation, we have that it contains all reachable states that satisfy $K\varphi$. Moreover, we know that $\sigma[n] \in [\![K\varphi]\!]$ iff $(\sigma, n) \models$

$K\varphi$. Due to the recursive definition of LTL semantics, we can replace the subformula with the symbolic characterization of the states that satisfy it. □

Using the above lemma, we can replace every epistemic subformula in ψ with its denotation, thus obtaining a pure LTL formula. In general, we are able to take a formula in KL_n and reduce it to a formula in KL_{n-1}, by removing one level of nesting. By applying this technique recursively we can reason on any KL formula.

Computing the denotation up-front is what we call *eager* approach. In general, this idea of delegating the handling of the epistemic atoms to an external oracle is not new, and it has been used in [152] (with the name *local proposition*), and in [153, 74] to deal with perfect recall. Existing approaches for KL_1 model-checking under observational semantics, rely on the availability of the denotation of the epistemic subformulae. In order to obtain the denotation, techniques based on BDDs first compute the reachable states and than partition the set of states based on their observability.

The starting points of those procedures is the computation of the reachable states. This computation is possible in the finite state case, and therefore this general algorithm is commonly implemented in tools for finite state temporal epistemic logic model-checking (both LTL and CTL) such as MCK [88] and MCMAS [116]. Unfortunately, in the context of infinite state systems, reachability is in general undecidable. Additionally, operations requiring quantifier elimination (such as backward images) on infinite state systems, range from expensive to undecidable.

11.2.1 Approximating the Denotation

Lemma 3 requires the computation of $[\![K\varphi]\!]$, and therefore the computation of the reachable states. This requirement can be relaxed by introducing

an approximation of the denotation called *good approximation*, that differs from the denotation only on non-reachable states.

Definition 30 (Good Approximation)**.** *Let P be a POTS and $K\varphi$ a KL_1 formula. We call a set of states $[\![K\varphi]\!]^*$ a* good approximation *of the denotation of $K\varphi$ iff $[\![K\varphi]\!] = [\![K\varphi]\!]^* \cap Reach_P$.*

Since the denotation is a subset of the reachables, it follows that $[\![K\varphi]\!] \subseteq [\![K\varphi]\!]^*$. In practice, we over-approximate the denotation by including states that are not reachable. The intuition is that we will let the LTL model-checking algorithm decide, at search-time, whether a given state in the approximation is reachable or not, and thus whether it belongs to the denotation.

Theorem 16. *Given a POTS P, and a KL_1 formula ψ s.t. $K\varphi$ is a subformula of ψ, we have that: $P \models \psi$ iff $P \models \psi[K\varphi / [\![K\varphi]\!]^*]$.*

Proof. Lemma 3 shows that we can replace the epistemic subformula with its denotation. We now need to show that we can replace the denotation with a good approximation.

Let us consider a trace σ of P and a point n, s.t. $(\sigma, n) \models K\varphi$ but $(\sigma, n) \not\models [\![K\varphi]\!]^*$. This means that $\sigma[n] \notin [\![K\varphi]\!]^*$, i.e., the state is not included in the good approximation. However, by definition of good approximation, we know that $[\![K\varphi]\!]^* \supseteq [\![K\varphi]\!]$ thus we reach a contradiction.

Let us consider the other direction, in which we have have a path σ of P and a point i s.t. $(\sigma, n) \not\models K\varphi$ but $(\sigma, n) \models [\![K\varphi]\!]^*$. By definition of good approximation, this means that $\sigma[n]$ must be non-reachable, thus reaching a contradiction. □

This result tells us that we can focus on an approximation. The approximation can be symbolically represented as an expression on the state variables of the system. However, we notice that due to the observational

semantics of K, whether a state s belongs to $[\![K\varphi]\!]$ depends only on the observation associate to the state ($obs(s)$) rather than the state itself. Therefore, we would like to represent the denotation using only the observable variables, thus obtaining a more compact representation of the denotation, and much better practical performances (as we will show Section 11.4).

Definition 31 (Observable Denotation)**.** *Given a denotation* $[\![K\varphi]\!]$*, we call* $obs([\![K\varphi]\!])$ *the* observable denotation *defined as:*

$$obs([\![K\varphi]\!]) = \{obs(s) \mid s \in [\![K\varphi]\!]\}$$

The observable denotation corresponds to an (over-approximating) abstraction of the set of states in $[\![K\varphi]\!]$, where we might have observations that belong to states that are both reachable and non-reachable. For example, consider two states s_1 and s_2, s.t. $obs(s_1) = obs(s_2)$ and s_1 is reachable but s_2 is not. Since s_1 is reachable, $obs(s_1)$ will be in the observable denotation, thus representing both states. This is in-line with the definition of good approximation and we can build a good approximation starting from the observable denotation. Recalling that $\Sigma(X)$ is the set of all possible assignments to the state variables X, we define the following lemma.

Lemma 4. *Let P be a POTS and $K\varphi$ be a KL_1 formula, then the observable denotation $obs([\![K\varphi]\!])$ is a good approximation of the denotation of $K\varphi$. Formally:*

$$[\![K\varphi]\!] = \{s \in \Sigma(X) \mid obs(s) \in obs([\![K\varphi]\!])\} \cap Reach_P$$

Proof. We need to show that all states that we are considering either belong to the denotation, or are not reachable.

- $s \in [\![K\varphi]\!]$: by definition $obs(s) \in obs([\![K\varphi]\!])$, therefore it belongs to the good approximation.

- $s \notin [\![K\varphi]\!]$: If there is a state s' with the same observation ($obs(s) = obs(s')$), then s' belongs to $[\![K\varphi]\!]^*$. Since by assumption it did not belong to the denotation, it means that either it is not reachable or it does not satisfy $K\varphi$. If it is not reachable, we are done due to the definition of good approximation. If it does not satisfy $K\varphi$, then neither s' satisfies $K\varphi$, since they have the same observation. Thus we reach a contradiction.

□

$obs([\![K\varphi]\!])$ provides a compact way of representing a good approximation of the denotation of $K\varphi$, that can be used to perform model-checking:

Theorem 17. *Given a POTS P and a formula ψ that contains an epistemic subformula $K\varphi$, the following holds:*

$$P \models \psi \text{ iff } P \models \psi[K\varphi/obs([\![K\varphi]\!])]$$

Proof. Theorem 16 shows that this holds for a good approximation, and Lemma 4 shows that the observable denotation is a good approximation. □

11.2.2 Parameter Synthesis

Given a state s, we can check whether it belongs to $[\![K_A\varphi]\!]$ by checking whether all points in traces with the same observation satisfy φ. This check can be encoded with the following LTL query:

$$P \models G((\bigwedge_{x \in O_A} x = s(x)) \rightarrow \varphi)$$

To compute the denotation, we need to find all the reachable states that satisfy the above property. By considering the state variables as parameters, we can see how this boils down to finding all those parameter values

that satisfy the property: a *parameter synthesis* problem (Definition 5). In many practical settings, introducing a parameter for each state variable could lead to a big parameter synthesis problem: e.g., if the system has 100 state variables, we need 100 parameters. We can thus introduce one parameter for each state variable (obtaining a good approximation) or we can introduce a parameter for each observable variable (obtaining an approximation of the observable denotation). Both approaches are correct, but we can obtain significant performance improvement by considering only the observable variables as parameters.

For each observable variable $o \in O_A$, we define a parameter $u_o \in U_O$ of the same type as o. This makes it possible to check whether we are in a state that is compatible with the observation ($\bigwedge_{o \in O_A}(o = u_o)$). A valuation of the parameters γ will describe exactly one observation that belongs to the denotation. We extend the original system $P = (V, I, T, O)$ by adding the parameters, obtaining $\tilde{P} = (V, U_O, I, T, O)$. The parameters do not have any direct impact on the transition relation, nor on the initial conditions. The relation between the state variables and the parameters is captured only by the property for which we perform the parameters synthesis problem:

$$\rho = \{\gamma \mid \tilde{P}_\gamma \models \gamma(G((\bigwedge_{o \in O_A} o = u_o) \rightarrow \varphi))\} =$$
$$\{\gamma \mid \tilde{P}_\gamma \models G((\bigwedge_{o \in O_A} o = \gamma(u_o)) \rightarrow \varphi)\}$$

Due to the shape of the property, the parameter synthesis problem might include observations that are not reachable. If an observation is not reachable, the left hand-side of the implication might be vacuously false, since we can never reach a state with the given observation. This means that the observation, although not reachable, will be part of the solution; in this case, we do not care about the relation between the state and the property

φ. This justifies the definition of a good approximation, and motivates Theorem 17. In order to show that this is a good approximation, we need to show that it matches Definition 30:

Lemma 5. *Let $\tilde{P}$ be the parametric extension of the POTS P, $K_A\varphi$ be a KL_1 formula and O_A the set of observable variables associated to the observer A. Let ρ be the result of the following parameter synthesis problem:*

$$\rho \;=\; \{\gamma \mid \tilde{P}_\gamma \models G((\bigwedge_{o \in O_A} o = \gamma(u_o)) \rightarrow \varphi)\}$$

then $[\![K_A\varphi]\!] = \{s \in \Sigma(X) \mid obs(s) \in \rho[u_o/o]\} \cap Reach$

Proof. In order to prove this result, we need to show that $\{s \in \Sigma(X) \mid obs(s) \in \rho[u_o/o]\}$ is a good approximation. We show that $\rho[u_o/o] \supset obs([\![K_A\varphi]\!])$, and in particular, that every observation that is present only in $\rho[u_o/o]$ represents a set of non-reachable states.

1. If $\overline{o} \in obs([\![K_A\varphi]\!])$ then $\overline{o} \in \rho[u_o/o]$: follows from the definition of parameter synthesis problem, and the fact that it is maximal, thus the set of parameters returned is the maximal set that satisfies the LTL property.

2. If $\overline{o} \in \rho[u_o/o] \setminus obs([\![K_A\varphi]\!])$ then $\forall s.obs_A(s) = \overline{o} \Rightarrow s \notin Reach$. Let us assume that s is reachable, but $\overline{o}$ is in $\rho[u_o/o] \setminus obs([\![K_A\varphi]\!])$. s must satisfy the LTL property $G((\bigwedge_{x \in O_A} x = s(x)) \rightarrow \varphi)$. This can be the case iff i) $\bigwedge_{x \in O_A} x = s(x)$ is never true, or ii) every reachable state with the same observation $\overline{o}$ satisfies φ. If i) is true, we reach a contradiction. If ii) is true, then $\overline{o}$ belongs to $obs([\![K_A\varphi]\!])$, thus reaching a contradiction.

□

The result of the lemma holds for both finite and infinite state systems. In the infinite state case, we are computing a region over the parameters that might be infinite, and the parameter synthesis engine might not terminate. In many practical cases, the observable variables of the system have finite domains. In these situations, the observable denotation is a finite set, that can be constructed by enumeration [109]. For finite state systems, this guarantees termination. However, for infinite state systems, this is not sufficient to guarantee termination, since the reachability problem for infinite state systems is undecidable in general [123], and thus the (LTL) underlying model-checking technique might not terminate. Notice that we are using parameter synthesis as a black-box, therefore we can use any off-the-shelf implementation. If φ is purely propositional, we can rewrite the above problem as an invariant parameter synthesis problem. This is relevant in practice since LTL parameter synthesis is a more difficult problem. Finally, to compute the good approximation on state variables, we only need to change the set of parameters, and do not need to modify the property.

11.3 Lazy Approach

Consider the property $G(K_A\beta \rightarrow \alpha)$: every-time that the observer knows β, then α holds. To disprove this property, eager approaches need to compute the denotation of $K_A\beta$ and then intersect it with the denotation of $\neg\alpha$. The intersection might represent only a small set of states (Figure 11.1). A lot of the computation performed up-front might not be needed (e.g., if α is always true). Moreover, in the case of infinite state systems, it might not be possible to represent the set of states of the denotation. For this reason, we develop the *lazy* approach, in which we compute only an approximation of the denotation that is sufficient to verify the property. This idea makes

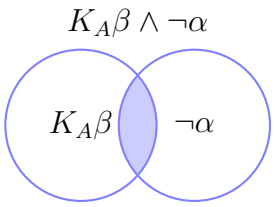

Figure 11.1: State-set intuition behind the Lazy approach.

it possible to better deal with infinite state systems, where it might not be possible to compute the reachable state set up-front.

Moreover, the technique is independent from the underlying model-checking algorithm, and thus we can apply modern verification techniques (e.g., IC3 [42]) and leverage the impressive advancements achieved in the formal verification community.

For a system to violate a property, we need a counter-example trace. We model-check an abstract version of the property, in which we treat all epistemic subformulas as propositional atoms. If we find a counter-example for this abstract property, we need to verify whether the counter-example satisfies all epistemic subformulas or whether it is spurious. If it is not spurious, we are done and the property is not satisfied. Otherwise, we need to refine our abstracted property, by learning additional constraints, and repeat. This flow is similar to the typical CEGAR [59] loop, with the significant difference that we do not refine the model but the property. The approach is divided into four main phases (Figure 11.2):

1. KL_1 to LTL abstraction
2. LTL Verification
3. Spuriousness check
4. Property Refinement.

```
1: function VERIFY(P, φ)
2:     φ_ρ, placeholders := BOOL_ABSTRACTION(φ)
3:     P_ρ := EXTEND(P, placeholders)
4:     loop
5:         cex := P_ρ ⊨ φ_ρ
6:         if not cex then
7:             return "Satisfied"
8:         end if
9:         if IS_SPURIOUS(P, cex, placeholders) then
10:            φ_ρ := LEARN_LEMMA(P, cex, placeholders, φ_ρ)
11:        else
12:            return cex
13:        end if
14:    end loop
15: end function
16:
17: function IS_SPURIOUS(P, cex, placeholders)
18:    for state ∈ cex do
19:        for ρ_{K_Aβ} ∈ placeholders do
20:            p_value := ρ_{K_Aβ}(state)
21:            if not ((state ∈ [[K_Aβ]]) ↔ p_value) then
22:                return True // Spurious!
23:            end if
24:        end for
25:    end for
26:    return False
27: end function
```

Figure 11.2: Lazy Algorithm Pseudo-Code

Property Abstraction

For every epistemic atom $K_A\beta$, we introduce a fresh *placeholder variable* $\rho_{K_A\beta}$ and obtain the abstracted property $\varphi_\rho = \varphi[K_A\beta/\rho_{K_A\beta}]$ by replacing each epistemic formula with the corresponding placeholder (Line 2 – BOOL_ABSTRACTION). The system is extended by adding the placeholder variables, that are initially unconstrained (Line 3). This corresponds to the most general abstraction of the property: in any state the placeholder

can be true or false. Counter-examples to the abstract property, represents assignments for the epistemic subformulas that can violate the property. For example, if a state of a counter-example contains a placeholder set to true, it means that in that state we want the epistemic subformula to hold. After this step, we have a property φ_ρ that is purely LTL, and an extended transition system (P_ρ) that contains all the variables of P plus the (unconstrained) placeholder variables.

LTL Verification

The main loop (Line 4-14) of the algorithm checks whether $P_\rho \models \varphi_\rho$. If we verify the abstract property on the system, then also the original property is satisfied. However, the converse is not true, due to spurious counter-examples, i.e., a counter-example that is not consistent from the epistemic point of view. For example, there is a state where the epistemic subformula holds, but that does not belong to the denotation of the epistemic formula. We iterate until a valid counter-example is found, or the property is shown to hold ($P_\rho \models \varphi_\rho$). If no counter-example is found (Lines 6-7), then the model satisfies the abstracted property, and therefore we can terminate: on some systems we are able to terminate without ever checking the epistemic part. If a counter-example exists, we need to check whether it is spurious (Line 9). If this is the case, we can exclude the counter-example, otherwise we have found a valid counter example.

Spuriousness Check and Refinement

To check the spuriousness of a counter-example, we need to check that each state of the counter-example satisfies the epistemic part. This is implemented in the function IS_SPURIOUS (Line 17). Each state can be checked in isolation because the transition relation is independent from the epistemic part. For each state of the counter-example, and for each

epistemic subformula (Lines 18-19), we extract the value of the placeholder in the current state (p_value, Line 20), and check whether the current state belongs to the denotation. If p_value is true but the state does not belong to the denotation or, viceversa, p_value is false and the state belongs to the denotation, then we are in a spurious state, and we can exclude the counter-example. If we validated all epistemic formulas in all the states, then the counter-example is a real counter-example (Line 26).

To know whether a state s belongs to $[\![K_A\beta]\!]$, ($s \in [\![K_A\beta]\!]$) we perform the following model-checking query:

$$P \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \beta) \tag{11.1}$$

Notice, that this is the same property that we use in the eager approach, when performing parameter synthesis. Instead of obtaining all states that satisfy the property (as in the eager approach) we check whether the current state satisfies it. Indeed, we ask whether each reachable state that has the same observation of s, satisfies the formula β. If P satisfies the property (positive case), we learn the lemma $\bigwedge_{x \in O} x = s(x) \rightarrow \rho_{K_A\beta}$, otherwise (negative case) we learn $\bigwedge_{x \in O} x = s(x) \rightarrow \neg\rho_{K_A\beta}$ (these are the lemmas returned by the LEARN_LEMMA function at Line 10). These lemmas impose additional constraints between all the states with the given observation and the placeholder variables, thus characterizing the epistemic atoms. In particular, if the counter-example is spurious, then we exclude it. Lemmas are learned by updating φ_ρ (Line 12), i.e., learned lemmas (λ_i) become preconditions to φ_ρ. Thus, in each iteration i, we have $\varphi_\rho^i := \lambda_i \rightarrow \varphi_\rho^{i-1}$.

$\boldsymbol{InvKL_1}$

If $\varphi \in InvKL_1$, the property rewriting step gives us a formula $G(\psi)$ where ψ is purely propositional. Since we are assuming deadlock freedom, this encodes an invariant over reachable states and we can take advantage of

this fact, by using ad-hoc reachability algorithms, instead of a full LTL algorithm. This provides us with a performance boost and, more importantly, guarantees us that the counter-example will be a finite trace ending in a state that violates ψ. During the validation of the counter-example we need to validate only the last state of the trace, thus obtaining significant speed-ups, since the validation phase does not depend anymore on the length of the counter-example traces.

Example

We apply the algorithm on the property: $G(K\beta \rightarrow \alpha)$. We first rewrite it as LTL, and introduce the placeholder variable: $\varphi_\rho = G(\rho_{K\beta} \rightarrow \alpha)$. The transition system $P = (V, I, T, O)$ is extended by adding the unconstrained placeholder variable $P_\rho = (V \cup \{\rho_{K\beta}\}, I, T, O)$. The main-loop of the procedure checks whether $P_\rho \models \varphi_\rho$. Let us assume that the answer is negative, and we get the trace:

$$(o_1, \alpha, \beta, \rho_{K\beta}), (\neg o_1, \neg\alpha, \beta, \rho_{K\beta})$$

that violates the abstract property, since the last state requires the epistemic formula to hold, but α does not hold. We check whether this is a spurious counter-example, by checking the consistency of each state of the trace. In particular, we want to know if the first state belongs to the denotation of $K\beta$, since the state has observation o_1, we verify whether : $P \models G(o_1 \rightarrow \beta)$. Let us assume that this query has a negative outcome, then we modify our property to include this lemma: $\varphi'_\rho = (o_1 \rightarrow \neg\rho_{K\beta}) \rightarrow \varphi_\rho$. Since this is a spurious state, the counter-example is considered spurious, and we need to find another counter-example. We check $P_\rho \models \varphi'_\rho$ and no counter-example is found. Thus, we conclude that $P \models G(K\beta \rightarrow \alpha)$.

11.3.1 Correctness and Termination

The following two lemmas show that the lazy approach is correct, i.e., $P \models \varphi$ iff the lazy algorithm terminates without a counter-example:

Lemma 6. *Given P and φ, and the associated P_ρ, φ_ρ, we have that if $P_\rho \models \varphi_\rho$ then $P \models \varphi$.*

Lemma 7. *Let σ_ρ be a trace of P_ρ and σ be its projection on the variables of P. If $P_\rho, \sigma_\rho \models \neg\varphi_\rho$ and, for all $n \geq 0$, for all $K_A\beta$ occurring in φ, $P_\rho, \sigma_\rho, n \models \rho_{K_A\beta}$ iff $\sigma[n] \in [\![K_A\beta]\!]_P$, then $P, \sigma \models \neg\varphi$.*

Lemma 7 requires that the counter-example is not spurious, and in particular, that we are able to check whether a state belongs to the denotation. Therefore, we need to prove that the model-checking query that we use to check whether a state belongs (or not) to the denotation is correct:

Theorem 18. *Given a reachable state s of P, the following three statements are equivalent:*

1. $s \in [\![K_A\beta]\!]_P$
2. $P \models G(\bigwedge_{x \in O_A} x = s(x) \rightarrow \beta)$
3. $P \models G(\bigwedge_{x \in O_A} x = s(x) \rightarrow K_A\beta)$

Proof. 1 $\Leftrightarrow$ 3) By the definition of KL_1, $P \models G(\bigwedge_{x \in O_A} x = s(x) \rightarrow \beta)$ iff for all σ, for all n, if $obs_A(\sigma[n]) = obs_A(s)$, then $(P, \sigma, n) \models K_A\beta$. Then, by the definition of $[\![K_A\beta]\!]_P$, the condition 1 holds iff s is reachable and 2 holds.

2 $\Rightarrow$ 3) Consider a trace σ and an integer n such that $obs_A(\sigma[n]) = obs_A(s)$. Since the condition 2 holds, then for all σ', for all n', if $obs_A(\sigma'[n']) = obs_A(s)$, then $(P, \sigma', n') \models \beta$; thus, for all σ', for all n', if $obs_A(\sigma'[n']) = obs_A(\sigma[n])$, then $(P, \sigma', n') \models \beta$. Thus, $(P, \sigma, n) \models K_A\beta$.

$3 \Rightarrow 2$) The condition 2 follows directly from 3 and the fact that $K_A\beta \rightarrow \beta$ is a tautology for all β (Axiom of knowledge **T**). □

Theorem 19. *Given a reachable state s of P, the following three statements are equivalent:*

1. $s \notin [\![K_A\beta]\!]_P$
2. $P \not\models G(\bigwedge_{x \in O_A} x = s(x) \rightarrow \beta)$
3. $P \models G(\bigwedge_{x \in O_A} x = s(x) \rightarrow \neg K_A\beta)$

Proof. $1 \Leftrightarrow 3$) follows from Theorem 18 by contraposition.

$2 \Rightarrow 3$) Consider a trace σ and an integer n such that $obs_A(\sigma[n]) = obs_A(s)$. Since the condition 2 holds, then there exists σ' and n' such that $obs_A(\sigma'[n']) = obs_A(s)$ and $(P, \sigma', n') \not\models \beta$; thus, it is not true that for all σ', for all n', if $obs_A(\sigma'[n']) = obs_A(\sigma[n])$, then $(P, \sigma', n') \models \beta$. Thus, $(P, \sigma, n) \models \neg K_A\beta$.

$3 \Rightarrow 2$) Suppose by contradiction that $P \models G(\bigwedge_{x \in O_A} x = s(x) \rightarrow \beta)$. Thus, by Theorem 18, $P \models G(\bigwedge_{x \in O_A} x = s(x) \rightarrow K_A\beta)$. Since s is reachable, then there exists σ and n such that $\sigma[n] = s$. Thus $(P, \sigma, n) \models K_A\beta$ and, by the condition 3, $(P, \sigma, n) \models \neg K_A\beta$, which is a contradiction. □

Finally, we need to show that the lemmas that we are adding to exclude a spurious counter-example are correct, i.e., the abstract model-checking problem is still a sound over-approximation of the concrete model-checking problem. More precisely, if we use the lemma to restrict the abstract state space and the abstract model-checking passes, then we can still conclude that $P \models \varphi$:

Theorem 20. *(Positive case) Assume that $P \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \beta)$ and $P_\rho \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \rho_{K\beta}) \rightarrow \varphi_\rho$ Then $P \models \varphi$.*

Proof. (Positive case) If $P_\rho \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \rho_{K\beta}) \rightarrow \varphi_\rho$, then i) $P \models G(\bigwedge_{x \in O} x = s(x) \rightarrow K\beta) \rightarrow \varphi$ (by Theorem 6). If $P \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \beta)$, then ii) $P \models G(\bigwedge_{x \in O} x = s(x) \rightarrow K\beta)$ (by Theorem 18). From i) and ii), we can deduce that $P \models \varphi$.

□

Theorem 21. *(Negative case) Assume that* $P \not\models G(\bigwedge_{x \in O} x = s(x) \rightarrow \beta)$ *and* $P_\rho \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \neg\rho_{K\beta}) \rightarrow \varphi_\rho$ *Then* $P \models \varphi$.

Proof. (Negative case) If $P_\rho \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \neg\rho_{K\beta}) \rightarrow \varphi_\rho$, then i) $P \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \neg K\beta) \rightarrow \varphi$ (by Theorem 6). If $P \not\models G(\bigwedge_{x \in O} x = s(x) \rightarrow \beta)$, then ii) $P \models G(\bigwedge_{x \in O} x = s(x) \rightarrow \neg K\beta)$ (by Theorem 19). From i) and ii), we can deduce that $P \models \varphi$. □

Termination

System	*InvKL*$_1$	*KL*$_1$
Finite	Complete	Complete
Infinite w/ Finite Domain Obs.	Relative Complete	Incomplete
Infinite w/ Infinite Domain Obs.	Incomplete	Incomplete

Table 11.1: Completeness

Our algorithm is guaranteed to terminate on finite state systems, since there is a finite number of possible values for observations and placeholders. Model checking for infinite state systems is in general undecidable, therefore we have the problem that the internal model-checking calls might not terminate. In practice, model-checkers for infinite state systems (e.g., UPPAAL [16], SAL [67], nuXmv [47]) can terminate on particular instances or classes of models. We call *relative complete* an algorithm that terminates assuming that all the model-checking queries terminate. Our approach is

relative complete for infinite state systems only if we have both finite domain observations, and finite counter-examples (as in the case of $InvKL_1$). Otherwise, the algorithm is incomplete, since we might need to enumerate infinitely many observations, or validate infinitely many states. Figure 11.1 summarizes the result. Despite the theoretical result, we will show in Section 11.4, that our approach is able in practice to verify many models of interest.

11.3.2 Optimizations

In order for the Lazy approach to be competitive, we need to minimize the number of queries that we perform to the model-checker, and be able to learn as much as possible from each query. The following are a few optimizations that we developed in order to achieve these objectives.

Static Learning

The placeholder variables are initially unconstrained. However, there are some facts that we can learn by looking at the property, for example, the axioms of epistemic logic. First, we know that $K\beta \rightarrow \beta$. This translates into the constraint: $\rho_{K\beta} \rightarrow \beta$. This ensures that we never need to validate a counter-example in which $\rho_{K\beta}$ and $\neg\beta$. Moreover, if o is a Boolean observable for the agent A, then $K_A o \leftrightarrow o$. Thus, we add the constraints $\rho_{Ko} \leftrightarrow o$ for each observable variable of the observer A.

Lemma Generalization

During refinement we learn something about a single observation. We generalize this to cover a bigger space of the observations, relating multiple observations to the value of the placeholder. For a state s and an observation o, we generalize the lemma $o \rightarrow \rho_{K\beta}$ into $o_1 \vee \cdots \vee o_n \rightarrow \rho_{K\beta}$ (similarly

for the negative case).

The main technique that we use to perform the generalization is parameter synthesis. Given the observation o, we remove some element, and perform the parameter synthesis starting from a partial assignment. We partition the set of observables in the ones we fix and the one we parameterize: $O = O_F \cup O_P$, $O_F \cap O_P = \emptyset$. For $K\beta$ we solve the following parameter synthesis problem:

$$\omega = \{o' \mid P \models G((\bigwedge_{x \in O_F} o(x) = x \wedge \bigwedge_{x \in O_P} o'(x) = x) \rightarrow \beta)\}$$

where o' is an assignment to the O_P variables. In practice, we obtain the region ω of assignments to O_P that (together with o) imply β. Since the region is maximal, all other assignments to O_P do not entail β. Therefore, we learn the lemma: $o \rightarrow (\omega \leftrightarrow \rho_{K\beta})$.

Choosing the partition into O_F and O_P is an interesting point of research, which we leave as future work. We propose a simple baseline heuristic, in which we randomly select a number w of observable variables. The choice of w is also heuristic, since a small value will cause overhead without gaining much generalization, while a big value will quickly lead to too many parameters. Indeed, picking $O_P = O$ (i.e., using all observable variables as parameters) is equivalent to solving the problem using the *eager* approach. In our implementation, we pick w as the logarithm of the iterations performed so far. In the future, we plan to study the impact of other heuristics like Luby series [118].

Dual-Rail Encoding

Validating a long counter-example is expensive. However, not all states might need to be validated. Let us consider the property: $a \wedge Xa \wedge XXKb$. A counter-example to this might be the trace:

$(a, b, \rho_{Kb}), (a, b, \neg\rho_{Kb}), (a, \neg b, \neg\rho_{Kb})$. The satisfaction of the epistemic subformula in the first two states is irrelevant. We would like the model-checker to identify those states so that we can skip them. Thus, we provide a way for the counter-example to contain *don't care* information, transforming the trace above to:

$$(a, b, -), (a, b, -), (a, \neg b, \neg\rho_{Kb})$$

This saves us from checking the first two states, and can be a significant saving when the traces are long. We use the Dual-Rail encoding [135] to encode three values: *True*, *False* and *Don't Care*. For a placeholder ρ_i we introduce the variables ρ_i^{True} and ρ_i^{False}, that are mutually exclusive. If ρ_i^{True} is true, then the placeholder is true; if ρ_i^{False} is true, then the placeholder is false; if both are false, then the placeholder has a do not care value. We then modify the IS_SPURIOUS function to handle the special case in which the placeholder is set to *don't care*, by considering the epistemic subformula as satisfied.

Positive Generalization via UNSAT-Cores

For epistemic atoms encoding safety properties, we can perform a more aggressive lemma generalization in case of a positive outcome from the placeholder query. If β is a safety property, when showing that $P \models G(o \rightarrow \beta)$ holds, an IC3-based model-checker will also provide us with an *inductive invariant* ι as witness. Since ι is an inductive invariant, $\iota \rightarrow (o \rightarrow \beta)$. We use unsat-core extraction to obtain a subset of the observations o that make the above unsatisfiable. In fact, $\iota \wedge o \wedge \neg\beta$ is unsatisfiable, and we obtain an unsat-core expressed over the observable variables that justifies the unsatisfiability [23]. Let us call o' such an unsat-core. By definition, we have that $\iota \wedge o' \rightarrow \beta$. Therefore, o' is a generalization of o. Moreover, the unsat core extraction is a purely combinational problem, that can be

efficiently handled by a SAT or SMT solver.

Voters Example

In the voters example (Section 11.1), there are infinitely many voting combinations that constitute a counter-example. The positive lemma UNSAT-Core generalization can help us quickly find the problem, by generating the lemma:

$$(result = 0 \wedge guess = 0) \rightarrow \rho_K$$

that justifies the counterexample. Let us assume that at least one voter did not vote 0. We rewrite the property as:

$$P \models G(voted \wedge (\bigvee_{v \in voter} vote_v \neq 0) \rightarrow \neg K_{jury} vote_1 = guess)$$

To show that no other counter-example exists, we would need to check all possible values of the votes, that are infinitely many. The negative generalization based on parameter synthesis allows us to terminate, by showing that any other voting is good.

11.3.3 Lazy Model-Checking of KL_n

Let us consider the voters example, and assume that there are only 2 voters. Each voter has access to its own vote, and to the total sum of the votes. In this case, the sum of the votes provides all the necessary information for each of the voters to know what the other voted. This follows from the fact that $result = vote_1 + vote_2$, thus if $vote_1 = 0$ then $vote_2 = result$.

$$G(K_1(vote_2 = (res - vote_1)) \wedge K_2(vote_1 = (res - vote_2)))$$

moreover, each voter knows that the other voter knows the vote. The simplest example is if $vote_1 = 0$:

$$G(vote_1 = 0 \rightarrow (K_1(vote_2 = res) \wedge K_1 K_2(vote_1 = 0)))$$

However, the jury does not know what each of them voted, assuming that one voted not zero and the other an arbitrary value p.

$$G(vote_1 \neq 0 \wedge vote_2 = p \rightarrow (\neg K_{jury}(vote_2 = p)))$$

To reason on this example, we need to extend the lazy approach to KL_n, and in particular to KL_2. This can be achieved by realizing that the lazy approach uses a solver for KL_{n-1} in order to reason about KL_n. For example, to reason about KL_1, we use an LTL (KL_0) model-checker. To check whether a state belongs to the denotation of $K\beta$ we use the query 11.1, that checks an LTL property over the original system. To extend the approach to KL_n, we need to perform a KL_{n-1} query on the system. For KL_2 we obtain:

$$P \models_{KL_1} G(\bigwedge_{x \in O} x = s(x) \rightarrow \beta) \tag{11.2}$$

The formula:

$$G(vote_1 = 0 \rightarrow (K_1(vote_2 = res) \wedge K_1 K_2(vote_1 = 0)))$$

is rewritten by introducing the following placeholder variables:

- $\rho_{K_1(vote_2=res)}$
- $\rho_{K_2(vote_1=0)}$
- $\rho_{K_1\rho_{K_2(vote_1=0)}}$

obtaining:

$$G(vote_1 = 0 \rightarrow (\rho_{K_1(vote_2=res)} \wedge \rho_{K_1\rho_{K_2(vote_1=0)}}))$$

During the spuriousness check, we will need to validate the value for the placeholder $\rho_{K_1\rho_{K_2(vote_1=0)}}$:

$$P \models_{KL_1} G(\bigwedge_{x \in O_1} x = s(x) \rightarrow K_2(vote_1 = 0))$$

Notice that we chose a rather simple β. However, we might have multiple epistemic and temporal operators, e.g., in the case of $K_1((FK_2p)\vee(GK_3p))$.

We are not aware of properties (in the literature or in practice) that require several levels of nesting (e.g., $n > 3$). Therefore, we believe that this recursive approach can be applied in practice. To achieve efficiency, however, optimizations need to be identified. In particular, we believe that a promising direction consists in the extension of static learning in order to account for axioms of relative knowledge. Exploring those optimizations is left as future work.

11.4 Experimental Analysis

A prototype implementation of our KL_1 model-checking algorithms was developed on top of on nuXmv [47] and related extensions for parameter synthesis using IC3 for infinite state systems [50].

We evaluated the scalability of the approach on infinite state models for both KL_1 and $InvKL_1$ properties, expressed over infinite state models using the theory of Linear Rational Arithmetic [73].

For the eager approach, we first show that expressing the parameter synthesis problem only on the observable variables (and not on all state variables) yields substantial performance gains. In this case, we report only the time required to perform the parameter synthesis step, since once the denotation is available, it is possible to apply any model-checking algorithm. In our case, we applied IC3, and the runtime of the verification part was negligible. The eager approach is then compared against existing BDD-based implementations, for systems in which there only few observables variables.

For the lazy approach, we compare the different optimizations to identify a competitive configuration. Once we identified a good configuration,

we compare it with the eager, and show that we can reach substantial performance improvements.

Finally, we compared our approach against the latest versions of two state-of-the-art model checkers: MCK [88][1] and MCMAS [116]. MCK implements various techniques for multiple types of temporal epistemic logic, not only limited to observational semantics. MCMAS implements BDD-based techniques for model-checking CTLK with observational semantics. CTLK is the epistemic extension of CTL. While the comparison with MCK is based on the same logic (KL_1), the comparison with MCMAS required us to translate the properties into CTLK. Nevertheless, the properties that we verified fall in the fragment that can be expressed in both logics.

11.4.1 Setup and Benchmarks

Experiments were executed on a 2.5Ghz Intel Xeon CPU, with a timeout of 1 hour, unless differently stated.[2]

The *Battery-Sensor* model (described in Section 4.5) encodes a typical subsystem found in aerospace designs, in which a set of redundant sensors are powered by a redundant power supply unit containing batteries that are modeled using real-valued variables. We study multiple properties related to faults in the system, for example:

$$G(fault\_gen_1 \rightarrow K fault\_gen_1)$$
$$G(K(gen_1.off \wedge gen_2.off) \rightarrow KX^{10}(\neg device.on))$$
$$G(fault\_psu \rightarrow FKO(fault\_psu))$$

The charge of the batteries is modeled using real-valued variables. We develop two models. In the first model, we assume that the diagnoser has a sensor that can provide the exact charge level of the battery. In

[1] We would like to thank Ron van der Meyden and Xiaowei Huang, for providing and supporting us with the use of an updated private version of MCK.

[2] Tools and Benchmarks are available at `http://marco.gario.org/phd/`.

common monitoring systems, however, the observers are rarely with infinite precision. Therefore, in the second model, the observations are simplified into three thresholds: low, mid and high (Bool Obs in Fig. 11.3). This is in-line with the type of monitoring defined in the PUS standard.

The second benchmark is a set of *magicboxes* [32], where a ball moves through a predefined pattern defined on a bi-dimensional grid. The external observer can only perform row and column observations, where row observation do not provide information on the column (and viceversa). The reasoner needs to consider the predefined path inside the magicbox and the available row/column information to try to identify the location of the ball. Scaling the size of the magicbox enables stressing the algorithms. For each magicbox we generate also an MCK and MCMAS model and test whether it is possible to know that the ball is in a given cell: $G(target\_cell \rightarrow Ktarget\_cell)$.

The *Dining Cryptographers* is a well studied problem in temporal epistemic logic [154]. A group of cryptographers gathered for a dinner and they are wondering if one of them paid the bill or whether the NSA paid. They devise a protocol to acquire this information, without the need of revealing the identity of the cryptographer that paid (if one did). We generated instances also for MCMAS[3] and MCK, for an increasing number of cryptographers (up to 400) and verify whether if one cryptographer paid, he knows that nobody else did:

$$G((done \wedge paid_1) \rightarrow K_1 \neg(\bigvee_{i \in [2..n]} paid_i))$$

and whether if a cryptographer paid, then it can eventually know that somebody else paid: $paid_1 \rightarrow XF(K_1 paid_2)$.

[3]We would like to thank Franco Raimondi, for providing us with a generator of DC problems for MCMAS.

11.4.2 Eager Approach

	Real Obs		Bool Obs	
Fault	State (32)	Obs (10)	State (32)	Obs (6)
Gen 1	**T.O.**	10.4	**T.O.**	1.9
Gen 2	**T.O.**	13.6	**T.O.**	2.1
Batt 1	**T.O.**	11.7	**T.O.**	1.3
Batt 2	**T.O.**	13.3	**T.O.**	1.5
Sens 1	**T.O.**	13.8	**T.O.**	1.7
Sens 2	**T.O.**	12.1	**T.O.**	1.9

Figure 11.3: Battery Sensor Runtime (seconds) and # of parameters in parenthesis.

Battery Sensor. Figure 11.3 reports the results of computing the denotation on the both the model with finite and infinite domain observations. The computation of the good approximation directly on the state variables cannot be concluded in this model under a time-out of 30 minutes. Instead, the observable denotation is computed in slightly more than 10s. The results for this case show that this type of problem can easily be handled with our technique using the observable denotation.

Magicbox. Figure 11.4 provides the count of variables used in each parameter synthesis problem. Figure 11.5 shows that the number of observables has a significant impact on both techniques. Intuitively, in the 60% case, the denotation is smaller. This benefits the observational denotation approach, while hinders the state denotation one. This is due to the fact that the parameter synthesis algorithm that we use works by complement: it needs to exclude many more states. Independently of the number of observables that we consider, the algorithm based on the state denotation always runs into time-out for the 20x20 case. To get a better idea of the relative performances we focus on a set of magicboxes of size between 10x10 and 19x19, with 90% and 80% of observability, and compare the runtime over 240 instances (Figure 11.6).

Benchmark Family	90%		80%		80%	
	State	Obs	State	Obs	State	Obs
10x10	22	18	22	16	22	12
20x20	42	36	42	32	42	24
30x30	62	54	62	58	62	36

Figure 11.4: Magicbox: # Parameters.

	90%		80%		60%	
Instance	State	Obs	State	Obs	State	Obs
10x10 Goal A	13.5	4.4	67.1	4.4	**T.O.**	3.8
10x10 Goal B	15.6	4.5	68.4	4.3	**T.O.**	3.7
10x10 Goal C	15.9	5.0	58.5	4.3	**T.O.**	4.0
20x20 Goal A	**T.O.**	81.0	**T.O.**	76.1	**T.O.**	50.3
20x20 Goal B	**T.O.**	83.3	**T.O.**	80.7	**T.O.**	52.1
20x20 Goal C	**T.O.**	80.7	**T.O.**	80.6	**T.O.**	52.3
30x30 Goal A	**T.O.**	544.0	**T.O.**	524.1	**T.O.**	344.9
30x30 Goal B	**T.O.**	579.3	**T.O.**	509.1	**T.O.**	387.6
30x30 Goal C	**T.O.**	578.3	**T.O.**	465.5	**T.O.**	367.3

Figure 11.5: Magicbox: Runtime (seconds).

Finite State. By considering the impact of the parameters on the the previous benchmarks, we expect our IC3-based parameter synthesis engine to perform well on problems of considerable size, as long as the number of the parameters is small. The comparison against BDD-based approaches implemented in MCK and MCMAS is carried out on a benchmark of finite-state magicboxes with 10% observable variables. Since the parameter synthesis engine is much more efficient in the finite case, we are able to significantly increase the size of the problems that we are able to handle when compared to infinite state models. Figure 11.7 shows the general trend of the runtime in seconds, where we can see that for this type of problems our parameter synthesis technique outperforms BDD-based approaches.

In summary, the experimental evaluation justifies the use of the observable denotation, instead of the naive approach of computing the denotation on state variables. Moreover, for problems in which the observable part is significantly smaller than the non-observable part, the eager technique can perform better than BDD-based approaches.

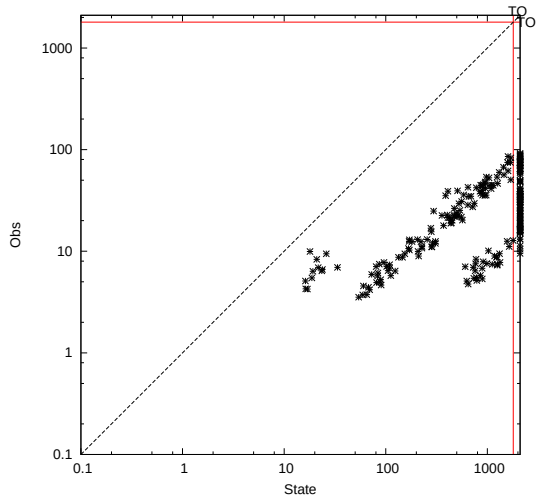

Figure 11.6: Magicbox: 10x10 to 19x19.

Size	Eager	MCK	MCMAS
100	23.88	106.59	1367.2
100	23.74	109.54	937.75
100	23.08	106.57	992.07
110	25.77	137.94	1723.66
110	25.56	149.46	1758.38
110	26.43	158.02	1410.05
120	31.24	234.27	2224.14
120	35.7	224.14	2293.25
120	33.42	186.47	2626.03

Size	Eager	MCK	MCMAS
130	38.96	258.41	2050.74
130	41.57	234.9	2206.66
130	43.24	276.71	3215.02
140	53.77	310.37	**T.O.**
140	48.98	282.5	2192.66
140	57.06	332.28	3178.17
150	64.48	408.09	**T.O.**
150	65.15	364.47	3443.91
150	65.73	411.45	3083.88

Figure 11.7: Comparison with MCK and MCMAS (TO: 3600s).

11.4.3 Lazy Approach

Optimizations Evaluation

To study the impact of all optimizations, we identified two main configurations and evaluated them on the complete benchmark set (Table 11.2). For full KL_1, we perform static learning, generalization and use the dual-rail encoding. For $InvKL_1$, we also perform static learning and generalization, but disable dual-rail encoding. Moreover, for $InvKL_1$, we only perform the validation of the last state of the trace.

To study the quality of these configurations, we proceed as follows. We

	Static Learning	Generalization	Dual-Rail	Last-State
KL_1	✓	✓	✓	✗
$InvKL_1$	✓	✓	✗	✓

Table 11.2: Lazy Configuration Summary

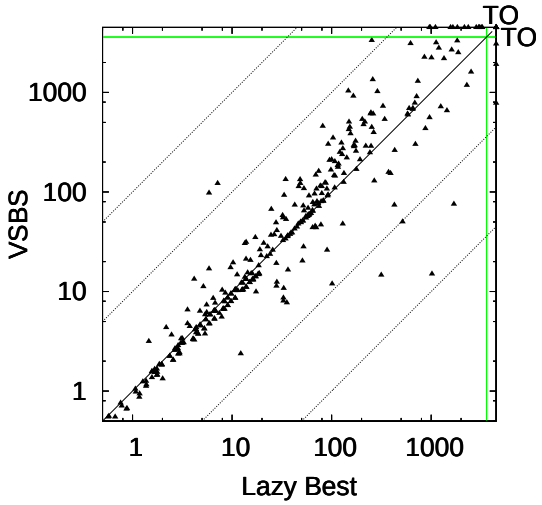

Figure 11.8: Comparing configurations.

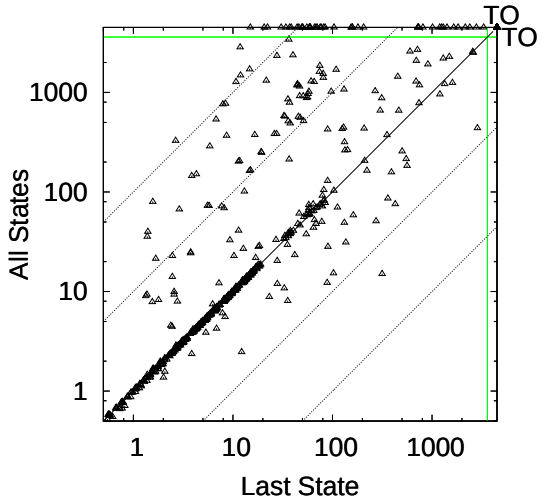

Figure 11.9: $InvKL_1$ Optimization.

compare our chosen configuration (Lazy Best) against all other possible configurations (i.e., for $InvKL_1$, we have 16 configurations in total). For each benchmark problem, we select the best configuration that is different from Lazy Best. We call that configuration Virtual Second Best Solver (VSBS). We compare Lazy Best and VSBS in Figure 11.8. Lazy Best times out in only 10 instances out of 349, while the VSBS times out in 20. Generalization sometimes adds significant overhead. We also compare the strategy of validating the whole trace vs. the last state (Figure 11.9): the latter can significantly pay off. The exceptions are cases in which a single counter-example is sufficient to learn everything needed to prove the property.

Generalization is fundamental to be able to solve the Dining Cryptographers benchmarks. In fact (Figure 11.10), applying generalization we are

#DC	Lazy w/o Gen.	Lazy w/ Gen.
3	0.26	0.15
5	1.01	0.15
7	6.07	0.13
9	58.03	0.15
11	1358.11	0.15
13	TO	0.15

Figure 11.10: DCs $InvKL_1$ properties (sec.)

Recall	#Obs	Lazy Basic	Lazy Best
0	11	3.28	1.46
5	66	3536.17	52.72
10	121	TO	103.47
20	231	TO	288.31
40	451	TO	981.11

Figure 11.11: Bounded Recall BS: Optimizations Impact (sec.)

able to solve all DC problems (up-to 400 cryptographers), while without generalization our approach times out when reaching 13 cryptographers.

For the infinite state benchmark (Battery Sensor), we considered problems with increasing bounded recall. Increasing the recall, increases the size of the model and the number of observables (Figure 11.11). Without any optimization, the algorithm times out with recall 6 (∼80 obs. variables), while with our chosen configuration, we can verify up to recall 40 (i.e., an infinite state model with 20 Real-valued variables and more than 500 Boolean variables).

Eager Approach

In Figure 11.12, we can see that the lazy approach scales better when increasing the problem size, i.e., for a recall of 5 the lazy approach terminates in less than a minute, while the eager approach reaches the timeout of 1 hour.

Finite State

In the finite case, we get excellent performances when compared to MCK and MCMAS. The comparison on all finite state problems (magicboxes and dining cryptographers) for MCK and MCMAS is given in Figures 11.13 and 11.14. In many cases our approach can provide up-to two orders of

Recall	Eager	Lazy
0	3.38	1.46
1	29.48	2.18
2	153.16	5.15
3	661.28	10.24
4	3028.94	13.44
5	T.O.	52.72

Figure 11.12: Bounded Recall BS: Eager vs Lazy (sec.)

magnitude improvement, and solves all the 118 instances, while MCMAS times out on 20, and MCK on 66. We highlight the results for the dining cryptographers benchmark in Figure 11.15, in which MCMAS is able to verify models only up-to 240 dining cryptographers (MCK is not included because it times-out at 20). The lazy approach can verify problem with 400 cryptographers in slightly more than 10 minutes.

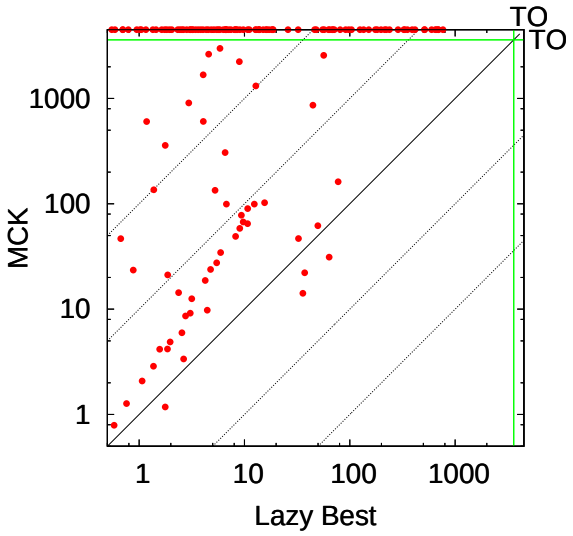

Figure 11.13: Lazy vs MCK (66/118 T.O.)

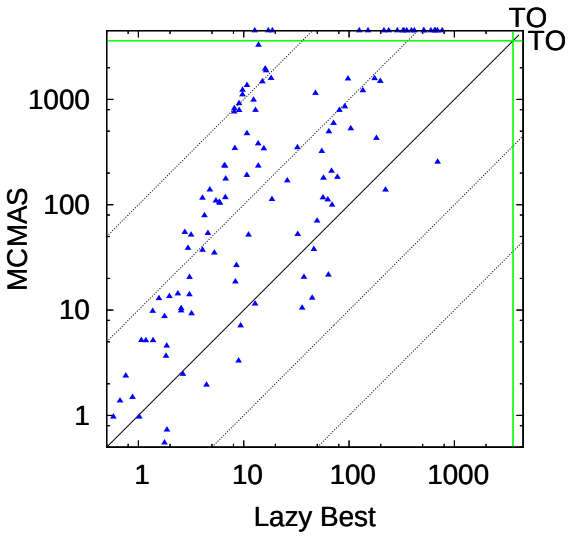

Figure 11.14: Lazy vs MCMAS (20/118 T.O.)

11.5 Chapter Summary

In this chapter, we presented two approaches for model checking the temporal epistemic logic KL_1 under observational semantics. These are the first approaches for KL_1 model checking over infinite state transition systems. We focus on KL_1 and on its fragment $InvKL_1$, because of their practical relevance both within this thesis and in the literature.

The eager approach is characterized by an up-front computation of the denotation of states satisfying the epistemic atom. This approach relies on the availability of a parameter synthesis engine as a black-box. Two variants of this approach were presented. First, we focus on the concept of *good approximation*, in order to characterize an over-approximation of the denotation expressed on state variables. Then, we present the *observable denotation* that is an over-approximation of the denotation expressed on the observable variables. Since the observable denotation is expressed on a smaller set of variables, it provides a simpler parameter synthesis problem.

The lazy approach is characterized by an on-demand computation of

#DC	MCMAS	Lazy
40	3.66	1.83
80	26.57	8.54
120	169.43	25.9
160	322.45	55.2
200	528.42	104.02
240	1582.68	174.86
280	T.O.	287.06
320	T.O.	391.57
360	T.O.	598.4
400	T.O.	765.96

Figure 11.15: DCs Runtime for KL_1 properties (sec.)

the pieces of the denotation that are needed to disprove a property. This approach relies on the availability of a model-checker as a black-box. The basic algorithm is described and exemplified. Moreover, we discuss several optimizations (i.e., static learning, generalization, and dual-rail encoding) that are important to achieve efficient reasoning.

In the experimental evaluation we compared these algorithms, against each other. Moreover, we compared the approach against state-of-the-art BDD-based model-checkers, showing that our approach is competitive, and allows reasoning over models that are out of reach for BDD-based engines.

There are several directions for future work, starting from the optimizations described in this Chapter that could be improved and extended. First, lemma generalization currently selects a random set of variables in order to perform the parameter synthesis problem. In the SAT community, several techniques and ideas have been explored to perform variable selection, e.g., variable activity. It would be interesting to study heuristics for this selection that can take into account, for example, how often the variable appears in the counter-examples. Second, the UNSAT-core generalization presented here works only for $InvKL_1$, since it relies on the inductive in-

variant obtained from IC3. Since the verification of KL_1 queries is done via K-Liveness [58], we do not obtain an inductive invariant. Therefore, the underlying model-checking algorithm should be extended accordingly. Third, bounded recall is currently handled by introducing new variables in the model. However, bounded recall is only relevant when checking spuriousness of the counter-example. In this case, we could extract the observations list from the counter-example, and check whether the series of observations belong to the denotation, by extending the query to the model-checker. This should allow us to keep the model small, even when considering long windows for multiple agents.

Finally, it would be interesting to evaluate whether this general technique could also be applicable to other types of modal logics, in which the truth of the modal subformula can be related to the state information, as we do here between epistemic expressions and observations.

Chapter 12

Conclusions and Future Work

This thesis provides a formal foundation for the design of FDI components based on temporal epistemic logic. This work has been motivated by the industrial need for a better process for FDI design. The theoretical aspects of the formalization have been shown together with their practical usage within the industrial projects.

Temporal epistemic logic is a suitable logic for the design of FDI components, thanks to its ability of describing reasoning about partially observable systems. This is also demonstrated by the close relation between FDI synthesis algorithm and TEL model-checking approaches.

Table 12.1 summarizes the relation between recall, design task, plant and associated reasoning technique. With the contribution of this thesis, it is possible to close the bounded recall case, in all its parts. For perfect-recall, we cover most of the process, but have some open points concerning synthesis and verification of infinite state systems. Table 12.2 provides a simplified overview of the results when limiting our specification to ASL (the fragment of ASL_K expressible in LTL – Table 6.1). Similarly, Table 12.3 shows a simplified overview when considering the whole ASL_K, stressing the fact that the open points concern only the combination of infinite state systems and perfect recall. In all these tables, the notation

"Abstraction" is meant to stress that, for example in the TFPG case, we can perform abstraction of the infinite state system and apply finite state techniques.

Table 12.1: Summary Table

Recall	Task	Plant	Sys. Diagnosable	Logic / Problem	Tool / Algorithm
BR	Diagnosability	Finite	No	KL_1	MCK/MCMAS/Lazy
			Yes	LTL	NuSMV
		Infinite	No	KL_1	Lazy
			Yes	LTL	nuXmv
	Verification	Finite	No	KL_1	MCK/MCMAS/Lazy
			Yes	LTL	NuSMV
		Infinite	No	KL_1	Lazy
			Yes	LTL	nuXmv
	Synthesis	Finite	No	Parameter Synthesis	nuXmv
			Yes	Parameter Synthesis	nuXmv
		Infinite	No	Parameter Synthesis	nuXmv
			Yes	Parameter Synthesis	nuXmv
PR	Diagnosability	Finite	No	KL_1	MCK
			Yes	LTL	NuSMV
		Infinite	No	KL_1	OPEN
			Yes	LTL	nuXmv
	Verification	Finite	No	KL_1	MCK
			Yes	LTL	NuSMV
		Infinite	No	KL_1	OPEN
			Yes	LTL	nuXmv
	Synthesis	Finite	No	Belief Explorer	xSAP
			Yes	Belief Explorer	xSAP
		Infinite	No	OPEN	OPEN/Abstraction
			Yes	OPEN	OPEN/Abstraction

12.1 Contributions

Formal characterization of properties of FDI Several key aspects of the FDI were formally characterized in terms of properties of an input/output transition system, that consumes observations and produces alarms. These properties include recall, correctness, completeness, delay, maximality, system and trace diagnosability, and context.

Table 12.2: Simplified Table for ASL

Recall	Task	Plant	Logic / Problem	Tool / Algorithm
0R	Diagnosability	Infinite	LTL	nuXmv
	Verification	Infinite	LTL	nuXmv
	Synthesis	Infinite	Parameter Synthesis	nuXmv
PR	Diagnosability	Infinite	LTL	nuXmv
	Verification	Infinite	LTL	nuXmv
	Synthesis	Finite	Belief Explorer	xSAP
		Infinite	OPEN	OPEN/Abstraction

Table 12.3: Simplified Table for ASL_K

Recall	Task	Plant	Logic / Problem	Tool / Algorithm
0R	Diagnosability	Infinite	KL_1	Lazy
	Verification	Infinite	KL_1	Lazy
	Synthesis	Infinite	Parameter Synthesis	nuXmv
PR	Diagnosability	Finite	KL_1	MCK
		Infinite	KL_1	OPEN
	Verification	Finite	KL_1	MCK
		Infinite	KL_1	OPEN
	Synthesis	Finite	Belief Explorer	xSAP
		Infinite	OPEN	OPEN/Abstraction

ASL_K The pattern-based Alarm Specification Language (ASL_K) was introduce to enable a simple way of specifying alarm conditions. Using the ASL_K semantics based on temporal epistemic logic, we are able to perform automated reasoning, including validation and verification, for different types of recall in a unified way.

Synthesis Algorithms for the synthesis of FDI components were presented. These algorithms are able to build an FDI that satisfy an ASL_K specification by construction. We presented algorithms for bounded-recall for both infinite and finite state systems. Moreover, we explained an approach for

perfect-recall for finite state systems, and explained the challenges related to extending this approach to infinite state systems.

Diagnosability The classical approach for diagnosability (i.e., the twin-plant approach) was extended in order to deal with different types of ASL_K specifications and recall. Using the new properties on the twin-plant, we are able to perform optimization of the sensors to be used by the diagnoser. We presented and evaluated an algorithm for Pareto Optimal Sensor Placement, where we want to optimize multiple cost functions at the same time.

Timed Failure Propagation Graphs Techniques for the validation of TFPGs based on SMT engines were presented and experimentally evaluated. We discussed also the use of the TFPGs as an abstraction of the system, in order to apply techniques for the finite state systems on timed systems.

Industrial Applications Many techniques presented in this thesis were used within the two ESA-funded projects AUTOGEF and FAME. We discussed the two projects and outline a general flow in which our formal approach can be applied.

***KL*$_1$ Model-Checking for infinite state systems** The first approach for model-checking of KL_1 with observational semantics over infinite state transition systems was presented and experimentally evaluated. A first version based on the eager computation of the denotation was presented and evaluated. We then showed how we do not need to perform the computation up-front, but can do it lazily. The lazy approach, together with several optimization, was presented and evaluated, showing excellent performances.

12.2 Future Work

Several interesting research directions open-up from here. In the domain of FDI design, an important step would be to apply our approach to more case-studies, in order to better identify potential limitations and bottlenecks of the process.

In this thesis we focused mainly on FDI. However, for the FDIR to be successful, it is important to consider the integration between FDI and FR, and provide a stronger foundation to FR specification and design. Although techniques have been developed and implemented in the planning and scheduling community, we believe that the main problem lies in how to properly specify what the FR can and should do. As an example, the concept of a recovery being successful is usually attached to the resolution of a potentially negative event. In a sense, the goal is not to reach a predefined target configuration, but it is to prevent something from happening. This is a slightly different approach than the one commonly adopted in the AI planning community.

Architectural Decomposition A concept that we did do not address is the architectural decomposition of the FDIR. Current FDIR design are stratified in multiple levels, depending on who is in charge of detecting and recovering from the faults. However, these levels are not commonly agreed upon, and their definition is usually more related to what they do rather than the objective they should achieve. Reasoning on a hierarchy of FDIR is an interesting challenge, that involves properly defining these concepts. The benefit, however, could be multiple. For example, by localizing the analysis to a sub-system, we could gain in performance during model-checking and synthesis. Additionally, new types of redundancy could be introduced, making the FDIR more robust and trustworthy.

Continuous Time Timed transition systems are a special type of infinite state transition systems. They have nice properties, and a vast literature exists dedicated to efficient algorithms and data-structures for reasoning on them. In this thesis, we only briefly touched the topic of timed behavior when discussing TFPGs and the FAME project. Timing aspects should be integrated better in the type of FDI that we want to build. This includes, for example, extending the underlying formalism to continuous time epistemic logic.

Prognosis In this thesis, alarms are associated to diagnosis conditions that talk about past events. By using diagnosis conditions that talk about the future events, one can imagine extending our framework to prognosis, i.e., the ability to anticipate the occurrence of a bad state. One main roadblocks when dealing with prognosis in this setting is the need to consider fairness constraints over the system model. Moreover, it is really difficult to predict something in a certain way. This suggests that, especially in the context of prognosis, we might be interested into moving into a quantitative setting, in which we can estimate the likelihood of reaching a bad state in the future.

Synthesis Algorithm The synthesis algorithm for perfect-recall on infinite state systems is an interesting direction of research and we discussed several ideas in Chapter 8. The same techniques used for synthesis can then be applied to model-checking. Techniques like abstraction, however, should not be limited to perfect-recall. Although we have technique for bounded-recall FDI synthesis, more work should be devoted in making those techniques more efficient. In turn, this will help to apply them to bigger and more complex designs.

Temporal Epistemic Models One of the main challenges that we faced while developing the algorithms for KL_1 model-checking was the limited availability of industrial models (and properties) using temporal epistemic logic. We believe this is due to two main factors. First, tools have different modeling languages, that are not always compatible. This requires a substantial amount of re-modeling. In this respect, MCK started supporting a symbolic format oriented towards symbolic transition systems. We believe that the existence of a common format can lead to drastic improvements (as demonstrated by both the SAT and SMT community). In particular, we are interested in foster the compatibility with SMV-like languages. Second, models tend to be released when they can be analyzed by the tools. Therefore, hard models tend not to be distributed. The same goes for specifications that cannot be reasoned upon. In particular, our decision to focus on KL_1 was in part motivated by its popularity in the benchmarks and the lack of general KL_n properties.

Lazy Temporal Epistemic Model Checking The tooling discussed in this thesis is a prototype, and a better integration with the nuXmv engine is planned in the future, in order to better integrate the temporal epistemic model-checking in the design process, and to leverage the continuous improvements in the area. In this direction, it would be interesting to explore possibilities to combine more tightly the IC3 algorithm with the lazy approach. In particular, several aspects of the IC3 algorithm could be extended, such as the use of generalized counter-examples, and the use of the approximation of the reachable states (kept internally by the engine) to decide epistemic atoms. On the Lazy algorithm itself, we plan to study additional techniques and heuristic for the generalization of counter-examples. By improving the performances for the KL_1 case, we expect to be able to better support KL_n in the future.

Bibliography

[1] Fides Aarts and Frits Vaandrager. Learning I/O automata. In *CONCUR 2010-Concurrency Theory*, pages 71–85. Springer, 2010.

[2] Martín Abadi and Leslie Lamport. Composing Specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[3] Sherif Abdelwahed, Gabor Karsai, and Gautam Biswas. System diagnosis using hybrid failure propagation graphs. Technical Report ISIS-02-302, Vanderbilt University, 2003.

[4] Sherif Abdelwahed, Gabor Karsai, Nagabhushan Mahadevan, and Stanley C Ofsthun. Practical implementation of diagnosis systems using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58(2):240–247, 2009.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[6] Les Atlas, George Bloor, Tom Brotherton, Larry Howard, Link Jaw, Greg Kacprzynski, Gabor Karsai, Ryan Mackey, Jay Mesick, Rick Reuter, and Mike Roemer. An evolvable tri-reasoner IVHM system. In *2001 IEEE Aerospace Conference, 11.0307*, Big Sky, Montana, USA, March 2001.

[7] AUTOGEF. Dependability Design Approach for Critical Flight Software. `https://es.fbk.eu/projects/autogef`.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[9] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, page 6. ACM, 2011.

[10] Pietro Baroni, Gianfranco Lamperti, Paolo Pogliano, and Marina Zanella. Diagnosis of large active systems. *Artificial Intelligence*, 110(1):135–183, 1999.

[11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[12] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

[13] Francesco Belardinelli, Davide Grossi, and Alessio Lomuscio. Finite abstractions for the verification of epistemic properties in open multi-agent systems. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 854–860. AAAI Press, 2015.

[14] Francesco Belardinelli, Andrew V Jones, and Alessio Lomuscio. Model checking temporal-epistemic logic using alternating tree automata. *Fundamenta Informaticae*, 112(1):19–37, 2011.

[15] Francesco Belardinelli, Alessio Lomuscio, and Fabio Patrizi. Verification of agent-based artifact systems. *J. Artif. Intell. Res. (JAIR)*, 51:333–376, 2014.

[16] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. *UPPAAL a tool suite for automatic verification of real-time systems*. Springer, 1996.

[17] E. Bensana, X. Pucel, and C. Seguin. Improving FDIR of spacecraft systems with advanced tools and concepts. In *Proc. ERTS*, 2014.

[18] Bernard Berthomieu and Miguel Menasche. An enumerative approach for analyzing time petri nets. In *Proceedings IFIP*. Citeseer, 1983.

[19] Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In Bernhard Nebel, editor, *IJCAI*, pages 473–478. Morgan Kaufmann, 2001.

[20] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, 2009.

[21] Pierre Bieber, Christian Bougnol, Charles Castel, Jean-Pierre Heckmann Christophe Kehren, Sylvain Metge, and Christel Seguin. Safety assessment with altarica. In *Building the Information Society*, pages 505–510. Springer, 2004.

[22] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[23] 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, 2014.

[24] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xSAP safety analysis platform. *Proceedings of TACAS 2016*, 2016.

[25] 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 *Proc. IMBSA 2014*, volume 8822 of *LNCS*, pages 82–95, 2014.

[26] Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, and Xavier Olive. Symbolic Synthesis of Observability Requirements for Diagnosability. In *AAAI*, 2012.

[27] 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)*, 2016.

[28] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νz-an optimizing smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015.

[29] J.-P. Blanquart and P. Valadeau. Model-based FDIR development and validation. In *Proc. MBSAW*, 2011.

[30] Ioana Boureanu, Mika Cohen, and Alessio Lomuscio. Automatic verification of temporal-epistemic properties of cryptographic protocols. *Journal of Applied Non-Classical Logics*, 2009.

[31] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Andrea Micheli. SMT-Based Validation of Timed Failure Propagation Graphs, 2015.

[32] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal Specification and Synthesis of FDI through an Example. In *Workshop on Principles of Diagnosis (DX'13)*, pages 174–179, 2013. Available at URL http://www.dx-2013.org/dx13-proceedings.pdf.

[33] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal Design of Fault Detection and Identification Components Using Temporal Epistemic Logic. In E. Ábrahám and K. Havelund, editors, *Proceedings of TACAS'14*, volume 8413 of *Lecture Notes in Computer Science*, pages 326–340, Grenoble, France, 2014. Springer.

[34] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal design of asynchronous FDI components using temporal epistemic logic. *Logical Methods in Computer Science*, 2015.

[35] Marco Bozzano, Alessandro Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability, and performance analysis of extended AADL models. *The Computer Journal*, doi: 10.1093/com, March 2010.

[36] 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.

[37] Marco Bozzano, Alessandro Cimatti, Oleg Lisagor, Cristian Mattarei, Sergio Mover, Marco Roveri, and Stefano Tonetta. Safety assessment of altarica models via symbolic model checking. *Science of Computer Programming*, 98:464–483, 2015.

[38] Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, and Stefano Tonetta. Formal safety assessment via contract-based design. In *Automated Technology for Verification and Analysis*, pages 81–97. Springer, 2014.

[39] Marco Bozzano, Alessandro Cimatti, Marco Roveri, Joost-Pieter Katoen, Viet Yen Nguyen, and Thomas Noll. Codesign of dependable systems: a component-based modeling language. In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, pages 121–130. IEEE Press, 2009.

[40] Marco Bozzano, Alessandro Cimatti, Marco Roveri, and Andrei Tchaltsev. A comprehensive approach to on-board autonomy verification and validation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011.

[41] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA*, pages 162–176. Springer, 2007.

[42] Aaron R Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[43] Aaron R Bradley, Zohar Manna, and Henny B Sipma. Whats decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006.

[44] Anibal Bregon, Matthew Daigle, Indranil Roychoudhury, Gautam Biswas, Xenofon Koutsoukos, and Belarmino Pulido. An event-based distributed diagnosis framework using structural model decomposition. *Artificial Intelligence*, 210:1–35, 2014.

[45] Vittorio Brusoni, Luca Console, Paolo Terenziani, and Daniele Theseider Dupré. A spectrum of definitions for temporal model-based diagnosis. *Artificial Intelligence*, 102(1):39–79, 1998.

[46] R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[47] 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.

[48] Elodie Chanthery and Yannick Pencolé. Monitoring and active diagnosis for discrete-event systems. In *7th IFAC symposium on fault detection, supervision and safety of technical systems*, pages 1545–1550, 2009.

[49] Alessandro Cimatti, Marco Gario, and Stefano Tonetta. A lazy approach to temporal epistemic logic model checking. In *15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS2016)*, 2016.

[50] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter synthesis with IC3. In *FMCAD*, pages 165–168. IEEE, 2013.

[51] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.

[52] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Verifying LTL properties of hybrid systems with k-liveness. In *Computer Aided Verification*, pages 424–440. Springer, 2014.

[53] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[54] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Solving strong controllability of temporal problems with uncertainty using SMT. *Constraints*, pages 1–29, 2014.

[55] Alessandro Cimatti, Charles Pecheur, and Roberto Cavada. Formal Verification of Diagnosability via Symbolic Model Checking. In *IJCAI*, pages 363–369, 2003.

[56] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. Validation of requirements for hybrid systems: A formal approach. *ACM Transactions on Software Engineering and Methodology*, 21(4):22, 2012.

[57] Alessandro Cimatti and Stefano Tonetta. Contracts-refinement proof system for component-based embedded systems. *Sci. Comput. Program.*, 97:333–348, 2015.

[58] Koen Claessen and Niklas Sorensson. A liveness checking algorithm that counts. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 52–59. IEEE, 2012.

[59] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[60] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE*, pages 321–330, 2011.

[61] The COMPASS Project. `http://compass.informatik.rwth-aachen.de`.

[62] FAME Consortium. D10: Fame prototype evaluation on a case study. Technical Report TASI-SD-FAM-ORP-0003, Issue 1, 2013.

[63] Marie-Odile Cordier, Philippe Dague, François Lévy, Jacky Montmain, Marcel Staroswiecki, and Louise Travé-Massuyès. Conflicts versus analytical redundancy relations: a comparative analysis of the model based diagnosis approach from the artificial intelligence and automatic control perspectives. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(5):2163–2177, 2004.

[64] Johan De Kleer, Alan K Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2):197–222, 1992.

[65] Johan De Kleer and Brian C Williams. Diagnosing multiple faults. *Artificial intelligence*, 32(1):97–130, 1987.

[66] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[67] Leonardo De Moura, Sam Owre, Harald Rueß, John Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. Sal 2. In *Computer aided verification*, pages 496–500. Springer, 2004.

[68] MCK Developers. *MCK 1.0.0: User Manual*, 2012.

[69] Cătălin Dima. Revisiting satisfiability and model-checking for ctlk with synchrony and perfect recall. In *Computational Logic in Multi-Agent Systems*, pages 117–131. Springer, 2009.

[70] Francois-Xavier Dormoy. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS08)*, pages 1–9, 2008.

[71] Abhishek Dubey, Gabor Karsai, and Nagabhusan Mahadevan. Model-based software health management for real-time systems. In *Aerospace Conference, 2011 IEEE*, pages 1–18, March 2011.

[72] 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 - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, pages 294–323, 2010.

[73] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for DPLL (T). In *Computer Aided Verification*, pages 81–94. Springer, 2006.

[74] Kai Engelhardt, Peter Gammie, and Ron Van Der Meyden. Model checking knowledge and linear time: Pspace cases. In *Logical Foundations of Computer Science*, pages 195–211. Springer, 2007.

[75] European Space Agency. ESTEC ITT AO/1-6992/11/NL/JK "FDIR Development and Verification & Validation Process", 2011.

[76] European Space Agency. Statement of Work: FDIR Development and Verification & Validation Process, 2011. Appendix to ESTEC ITT AO/1-6992/11/NL/JK.

[77] European Space Agency. Statement of Work: Hardware-Software Dependability for Launchers, 2012. Appendix to ESTEC ITT AO/1-7263/12/NL/AK.

[78] Ronald Fagin, Yoram Moses, Joseph Y Halpern, and Moshe Y Vardi. *Reasoning about knowledge*. MIT press, 2003.

[79] FAME. FDIR Development and Verification & Validation Process. `https://es.fbk.eu/projects/fame`.

[80] HP Feiler, B Lewis, and S Vestal. The SAE architecture analysis and design language (AADL) standard. In *IEEE RTAS Workshop*, 2003.

[81] P Feiler. Architecture analysis and design language (AADL) annex volume 3: Annex E: Error model v2 annex. *Number SAE AS5506/3 (Draft) in SAE Aerospace Standard. SAE International*, 2013.

[82] 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.

[83] Alexander Feldman. *Hierarchical approach to fault diagnosis*. PhD thesis, Masters thesis, Delft University of Technology, 2004.

[84] Alexander Feldman, Gregory Provan, and Arjan Van Gemund. A model-based active testing approach to sequential diagnosis. *Journal of Artificial Intelligence Research*, 39:301, 2010.

[85] European Cooperation for Space Standardization. ECSS-E-ST-70-11C: Space segment operability. Technical report, 2008.

[86] European Cooperation for Space Standardization. ECSS-Q-ST-30-11C:space product assurance. Technical report, 2011.

[87] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In Armin Biere and Carla Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer Berlin Heidelberg, 2006.

[88] Peter Gammie and Ron van der Meyden. Mck: Model checking the logic of knowledge. *Computer Aided Verification*, pages 256–259, 2004.

[89] Marco Gario and Andrea Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT-Workshop*, 2015.

[90] Fausto Giunchiglia and Chiara Ghidini. Local models semantics, or contextual reasoning = locality + compatibility. In *KR'98*, pages 282–291. Morgan Kaufmann, 1998.

[91] Alban Grastien and Gianluca Torta. A theory of abstraction for diagnosis of discrete-event systems. In Michael R. Genesereth and Peter Z. Revesz, editors, *Proceedings of the Ninth Symposium on Abstraction, Reformulation, and Approximation, SARA 2011, Parador de Cardona, Cardona, Catalonia, Spain, July 17-18, 2011.* AAAI, 2011.

[92] J. Greenyer, A.M. Sharifloo, M. Cordy, and P. Heymans. Efficient consistency checking of scenario-based product-line specifications. In *Requirements Engineering Conference (RE)*, pages 161–170, 2012.

[93] Joseph Halpern and Moshe Vardi. The complexity of reasoning about knowledge and time. lower bounds. *Journal of Computer and System Sciences*, 38(1):195–237, 1989.

[94] Joseph Y Halpern, Ron Van Der Meyden, and Moshe Y Vardi. Complete axiomatizations for reasoning about knowledge and time. *SIAM Journal on Computing*, 33(3):674–703, 2004.

[95] Zyad Hassan, Aaron R Bradley, and Fabio Somenzi. Incremental, inductive ctl model checking. In *Computer Aided Verification*, pages 532–547. Springer, 2012.

[96] Sandra Hayden, Nikunj Oza, Robert Mah, Ryan Mackey, Sriram Narasimhan, Gabor Karsai, Scott Poll, Somnath Deb, and Mark Shirley. Diagnostic Technology Evaluation Report For On-Board Crew Launch Vehicle. (September), 2006.

[97] Jinbo Huang and Adnan Darwiche. On compiling system models for faster and more scalable diagnosis. In *Proceedings Of The National Conference On Artificial Intelligence*, volume 20, page 300, 2005.

[98] Xiaowei Huang and Ron van der Meyden. The complexity of epistemic model checking: Clock semantics and branching time. In *ECAI*, pages 549–554, 2010.

[99] Wojciech Jamroga and Wiebe van der Hoek. Agents that know how to play. *Fundamenta Informaticae*, 63(2-3):185–220, 2004.

[100] Thierry Jéron, Hervé Marchand, Sophie Pinchinat, and Marie-Odile Cordier. Supervision patterns in discrete event systems diagnosis. In *Discrete Event Systems, 2006 8th International Workshop on*, pages 262–268. IEEE, 2006.

[101] Shengbing Jiang and Ratnesh Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications. In *IEEE Transactions on Automatic Control*, pages 128–133, 2001.

[102] Priscilla Kan John and Alban Grastien. Local consistency and junction tree for diagnosis of discrete-event systems. In *ECAI*, pages 209–213, 2008.

[103] Magdalena Kacprzak, Alessio Lomuscio, and Wojciech Penczek. Verification of multiagent systems via unbounded model checking. In *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*, pages 638–645. IEEE, 2004.

[104] Gabor Karsai. FACT: Fault adaptive control technology tools for developing fault adaptive systems. Tool presentation at `https://fact.isis.vanderbilt.edu/FACT-Sum.pdf`, 2013.

[105] Gabor Karsai, Sherif Abdelwahed, and Gautam Biswas. Integrated diagnosis and control for hybrid dynamic systems. In *AIAA Guidance, Navigation and Control Conference*, 2003.

[106] S.C. Kleene. *Mathematical Logic*. J. Wiley & Sons, 1967.

[107] Filippos Kominis and Hector Geffner. Beliefs in multiagent planning: From one agent to many. In *Proc. ICAPS Workshop on Distributed and Multi-Agent Planning*, 2014.

[108] Marta Kwiatkowska, Alessio Lomuscio, and Hongyang Qu. Parallel model checking for temporal epistemic logic. In *European Conference on Artificial Intelligence*, 2010.

[109] Shuvendu K Lahiri, Robert Nieuwenhuis, and Albert Oliveras. Smt techniques for fast predicate abstraction. In *Computer Aided Verification*, pages 424–437. Springer, 2006.

[110] Gianfranco Lamperti and Marina Zanella. Diagnosis of discrete-event systems from uncertain temporal observations. *Artificial Intelligence*, 137(1):91–163, 2002.

[111] Gianfranco Lamperti and Marina Zanella. Context-sensitive diagnosis of discrete-event systems. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 969, 2011.

[112] Francois Laroussinie, Nicolas Markey, and Philippe Schnoebelen. Temporal logic with forgettable past. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 383–392. IEEE Computer Society, 2002.

[113] Xavier Le Guillou, Marie-Odile Cordier, Sophie Robin, and Laurence Rozé. Chronicles for on-line diagnosis of distributed systems. In *Proceedings of the 2008 conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pages 194–198. IOS Press, 2008.

[114] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193, pages 196–218. Springer Berlin Heidelberg, 1985.

[115] Alessio Lomuscio and Jakub Michaliszyn. An abstraction technique for the verification of multi-agent systems against atl specifications. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, pages 428–437, 2014.

[116] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas: an open-source model checker for the verification of multi-agent sys-

tems. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2015.

[117] Alessio Lomuscio and Franco Raimondi. The complexity of model checking concurrent programs against ctlk specifications. In *Declarative Agent Languages and Technologies IV*, pages 29–42. Springer, 2006.

[118] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, pages 128–133. IEEE, 1993.

[119] Yunosuke Maki, Kenneth Loparo, et al. A neural-network approach to fault detection and diagnosis in industrial processes. *Control Systems Technology, IEEE Transactions on*, 5(6):529–541, 1997.

[120] A. Martinez Barrio, M. Verhoef, and J.L. Terraillon. FDIR: state of affairs. In *9th ESA Workshop on Avionics, Data, Control and Software Systems - ADCSS 2015*, 2015.

[121] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[122] Artur Meski, Wojciech Penczek, Maciej Szreter, Bozena Woznа-Szczesniak, and Andrzej Zbrzezny. Two approaches to bounded model checking for linear time logic with knowledge. In *Agent and Multi-Agent Systems. Technologies and Applications*, pages 514–523. Springer, 2012.

[123] Marvin L Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

[124] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 243–257. Springer, 2008.

[125] Gilles Mourot, S Bousghiri, and Jose Ragot. Pattern recognition for diagnosis of technological systems: a review. In *Systems, Man and Cybernetics, 1993.'Systems Engineering in the Service of Humans', Conference Proceedings., International Conference on*, pages 275–281. IEEE, 1993.

[126] Nicola Muscettola, P Pandurang Nayak, Barney Pell, and Brian C Williams. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1):5–47, 1998.

[127] 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.

[128] NASA. Fault management handbook (draft 2). Technical report, 2012.

[129] Description of On-Board Monitoring Service. `http://spd-web.terma.com/Projects/OBOSS/Home_Page/id109.htm`.

[130] Special Committee 205 of RTCA. *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[131] Stanley Ofsthun and Sherif Abdelwahed. Practical applications of timed failure propagation graphs for vehicle diagnosis. In *Autotestcon, 2007 IEEE*, pages 250–259, 2007.

[132] A. Oganessian. FDIR engineering supported by ECSS. In *5th ESA Workshop on Avionics, Data, Control and Software Systems - ADCSS 2011*, 2011.

[133] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[134] Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

[135] Jan-Willem Roorda and Koen Claessen. SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In *Computer Aided Verification*, pages 175–189. Springer, 2006.

[136] SAE. Arp4761 guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *SAE International, December*, 1996.

[137] Meera Sampath, Raja Sengupta, Stephane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9), 1995.

[138] 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.

[139] Monika Schubert, Alexander Felfernig, and Monika Mandl. Fastxplain: Conflict detection for constraint-based recommendation problems. In *Trends in Applied Intelligent Systems*, pages 621–630. Springer, 2010.

[140] Anika Schumann. Diagnosis of discrete-event systems using binary decision diagrams. *Workshop on Principles of Diagnosis (DX'04)*, pages 197–202, 2004.

[141] Anika Schumann, Yannick Pencolé, and Sylvie Thiébaux. A decentralised symbolic diagnosis approach. In *ECAI*, pages 99–104, 2010.

[142] Johann M Schumann, Timmy Mbaya, and Ole J Mengshoel. Bayesian software health management for aircraft guidance, navigation, and control. In *Annual conference of the prognostics and health management society 2011 (PHM-11)*, 2011.

[143] Roberto Sebastiani and Patrick Trentin. Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 335–349. Springer, 2015.

[144] David E Smith and Daniel S Weld. Conformant graphplan. In *AAAI/IAAI*, pages 889–896, 1998.

[145] Ashok Narain Srivastava and Jiawei Han, editors. *Machine learning and knowledge discovery for engineering systems health management.* Chapman & Hall/CRC data mining and knowledge discovery series. CRC Press, Boca Raton, FL, 2012.

[146] Roni Tzvi Stern, Meir Kalech, Alexander Feldman, and Gregory M Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI*, volume 12, pages 828–834, 2012.

[147] Shane Strasser and John Sheppard. Diagnostic alarm sequence maturation in timed failure propagation graphs. In *Autotestcon*, 2011.

[148] Xingyu Su, Alban Grastien, and Yannick Pencolé. Window-based diagnostic algorithms for discrete event systems: What information

to remember. In *The 25th International Workshop on Principle of Diagnosis (DX 2014)*, 2014.

[149] David Thorsley and Demosthenis Teneketzis. Diagnosability of stochastic discrete-event systems. *Automatic Control, IEEE Transactions on*, 50(4):476–492, 2005.

[150] Gianluca Torta and Pietro Torasso. On the use of OBDDs in model-based diagnosis: An approach based on the partition of the model. *Knowledge-Based Systems*, 19(5):316–323, 2006.

[151] Stavros Tripakis. Fault diagnosis for timed automata. In *Formal techniques in real-time and fault-tolerant systems*, pages 205–221. Springer, 2002.

[152] Wiebe Van Der Hoek, Michael Wooldridge, and Sieuwert van Otterloo. Model checking knowledge and time via local propositions: Cooperative and adversarial systems. 2004.

[153] Ron van der Meyden and Nikolay V Shilov. Model checking knowledge and time in systems with perfect recall. In *Foundations of Software Technology and Theoretical Computer Science*, pages 432–445. Springer, 1999.

[154] Ron Van der Meyden and Kaile Su. Symbolic model checking the knowledge of the dining cryptographers. In *17th IEEE Computer Security Foundations Workshop*, 2004.

[155] Venkat Venkatasubramanian, Raghunathan Rengaswamy, Surya N Kavuri, and Kewen Yin. A review of process fault detection and diagnosis: Part iii: Process history based methods. *Computers & chemical engineering*, 27(3):327–346, 2003.

[156] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault tree handbook. Technical report, DTIC Document, 1981.

[157] Bozena Woźna, Alessio Lomuscio, and Wojciech Penczek. Bounded model checking for knowledge and real time. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 165–172. ACM, 2005.

[158] xSAP: eXtended Safety Analysis Platform. `http://es.fbk.eu/tools/xsap`.