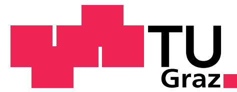

Edi Muškardin, univ. bacc. inf.

A Framework for Model-Based Diagnosis of Cyber-Physical Systems

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Co-Supervisor

Dipl.-Ing. Dr.techn. Ingo Hans Pill

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Graz, June 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

Everyday reliance on cyber-physical systems requires them to be fault-tolerant, self-adapting, and independent of human interaction. However, faults are unavoidable, and techniques like model-based diagnosis help in locating and dealing with faults and their consequences. Cyber-physical systems are characterized by complex interactions between their individual cyber and physical components, and their exhaustive testing is often impossible due to the input size.

The model-based diagnosis uses a model of the system and observations collected at run time to reason about the correctness of the system and its components. This thesis introduces a framework that eases the task of creating and testing models used for diagnosis purposes.

Employing an interface to the modeling language Modelica, a designer can simulate a cyber-physical system's detailed behavior, and based on the observed data then assesses the diagnostic solution(s) under development and explore the trade-offs of individual solutions. Alongside the diagnosis model, the special interface enables the designer to develop a controller that can be used to explore the effect of compensating or repair actions at run time. Part of the thesis is dedicated to the automatic generation of abductive models using simulation and fault injection. The automatic generation of models is an appealing way to bridge a gap between modeling costs and diagnostic benefits, as the gap is often too wide in real-world scenarios.

CatIO is an abbreviation of a Latin phrase "**Ca**usarum Cogni**tio**" meaning *(seek) knowledge of causes* and with CatIO the tedious process of finding causes of faults is made easier.

Abstract

Die alltägliche Abhängigkeit von cyber-physikalischen Systemen erfordert, dass sie fehlertolerant, selbstanpassend und unabhängig von menschlicher Interaktion sind. Fehler sind jedoch unvermeidlich, und Techniken wie die modellbasierte Diagnose helfen bei der Lokalisierung und Behandlung von Fehlern und deren Folgen. Cyber-physikalische Systeme zeichnen sich durch komplexe Interaktionen zwischen ihren einzelnen Cyber- und physikalischen Komponenten aus, und ihre erschöpfende Prüfung ist aufgrund der Größe des Inputs oft unmöglich.

Die modellbasierte Diagnose verwendet ein Modell des Systems und Beobachtungen, die zur Laufzeit gesammelt werden, um über die Korrektheit des Systems und seiner Komponenten zu argumentieren. Diese Arbeit stellt einen Framework vor, der das Erstellen und Testen von Modellen für Diagnosezwecke erleichtert.

Mithilfe einer Schnittstelle zur Modellierungssprache Modelica kann ein Konstrukteur das detaillierte Verhalten eines cyber-physikalischen Systems simulieren und auf der Grundlage der beobachteten Daten dann die in Entwicklung befindliche(n) Diagnoselösung(en) bewerten und die Kompromisse einzelner Lösungen erkunden. Neben dem Diagnosemodell ermöglicht die spezielle Schnittstelle dem Designer die Entwicklung eines Reglers, mit dem die Auswirkungen von Kompensations- oder Reparaturaktionen zur Laufzeit untersucht werden können. Ein Teil der Dissertation ist der automatischen Generierung abduktiver Modelle mittels Simulation und Fehlerinjektion gewidmet. Die automatische Generierung von Modellen ist ein ansprechender Weg, um eine Lücke zwischen Modellierungskosten und diagnostischen Vorteilen zu überbrücken, da die Lücke in realen Szenarien oft zu groß ist.

CatIO ist eine Abkürzung des lateinischen Ausdrucks "**Ca**usarum Cogni**tio**", was *(suchen) Wissen über Ursachen* bedeutet, und CatIO erleichtert den mühsamen Prozess der Fehlerursachenfindung.

Contents

Abstract iii

1. Introduction 1
- 1.1. Motivation 1
- 1.2. Objectives and Scope 3
- 1.3. Structure 4
- 1.4. Running Example 4

2. From Logic to Diagnosis 7
- 2.1. Logic 8
- 2.2. From Logic to Model 8
- 2.3. From Model to Diagnosis 10
- 2.4. Weak and Strong Fault Models 13
- 2.5. Consistency Based Diagnosis 14
  - 2.5.1. Unsatisfiable Subset Extraction 16
  - 2.5.2. RC-Tree 17
- 2.6. Abductive Diagnosis 19
  - 2.6.1. Assumption Based Truth Maintenance System 20
- 2.7. Mixed level covering arrays 21
- 2.8. Models and Simulation 21

3. Design and Implementation of a Diagnosis Framework 25
- 3.1. Workflow 25
- 3.2. Front end 29
  - 3.2.1. Modelica Models 29
  - 3.2.2. Diagnosis Models 33
  - 3.2.3. Interfaces 37
  - 3.2.4. Drivers 41

3.3. Back end . . . 45
3.3.1. Submodules . . . 45
3.4. Graphical User Interface . . . 50
3.4.1. Data extraction and scenario creators . . . 50
3.4.2. Modeling . . . 52
3.5. Code and Building/Running Instructions . . . 56

4. Usage and Examples **59**
4.1. Modeling the Differential Drive Robot . . . 59
4.1.1. Modeling in Modelica . . . 59
4.1.2. Diagnosis and Repair . . . 62
4.2. Automatic generation of abductive model . . . 69
4.2.1. Differential Drive Robot . . . 69
4.2.2. Automatic generation of RC-Circuits of various sizes . 73

5. Conclusion and Future Work **79**
5.1. Conclusion . . . 79
5.2. Future Work . . . 80

A. Examples **85**
A.1. ModelData from JSON file . . . 85
A.2. Simulation Scenario defined in JSON . . . 87
A.3. Diagnosis and Repair Example . . . 89
A.4. Circuit Encoder and Diff Implementation . . . 95

Bibliography **97**

List of Figures

1.1. A mobile robot with differential drive 5

2.1. And gate with inputs I1 and I2 and output O1 9
2.2. Model based diagnosis 12

3.1. CatIO's arhitecture 26
3.2. CatIO's inputs and outputs 26
3.3. CatIO's high level module interactions 27
3.4. ISCAS Benchmark Circuit c17 34
3.5. Encoding of persistent and intermittent faults 43
3.6. Output of different consistency based diagnosis procedures . 44
3.7. Data extraction and simulation scenario creator window . . . 51
3.8. Mixed level covering array generator window 53
3.9. Consistency based modeling window. 54
3.10. Abductive modeling window 55
3.11. CatIO's directory/file structure 57

4.1. Modelica model in connection editor 60
4.2. Robot movement without any repair or compensating action 64
4.3. Robot movement with repair action 65
4.4. Robot movement with compensating action 67
4.5. Robot movement with repair and compensating actions . . . 68
4.6. Automatically generated simulation scenarios. 69
4.7. Behavior of the simple electric circuits over time. 74
4.8. Schematics of the simple electric circuits. 75

A.1. Extraction of data. 89

Acknowledgments

Firstly I would like to thank my supervisors Univ.-Prof. Dipl.-Ing. Dr.techn. Fraz Wotawa and Dipl.-Ing. Dr.techn. Ingo Pill for their willingness to answer my questions, as well as for providing me with the opportunity to work in *Quality Assurance Laboratory for Autonomous Cyber-Physical Systems*, where I was able to develop the framework presented in this thesis. Both supervisors allowed me to experiment with my ideas and guided me in the development of the framework.

My colleges at the Institute of Software Technology also help me with advice regarding the thesis itself and other work, so I would like to take this opportunity to acknowledge their help and thank them.

Finally, biggest thanks goes to my parents, who help me achieve my goals throughout my life and education, and the level of their support cannot be expressed in words.
Special thanks goes to my grandma, *Nona Paola*, as she was the only one who always understood the sorrows of student life.

1. Introduction

In the first section, we will outline the real-world motivation behind the need for a framework like CatIO and the desired outcomes of the framework. The outline of the thesis structure is presented, as well as a running example which we will use to demonstrate certain concepts through the thesis.

1.1. Motivation

Nowadays, intelligent cyber-physical systems (*CPS*) are deeply integrated into everyday life and with future developments of artificial intelligence and the internet of things [25], our reliance on their ability to perform tasks autonomously will continue to grow. For an intelligent cyber-physical system, it is crucial to be aware of its current status and also that of its environment. This knowledge is used to make the right choices when having to deal with issues like faults. However, in cyber-physical systems interconnections of the environment and the cyber system itself makes them very complex and introduce several challenges in areas such as testing, verification, and validation, as well in model-based diagnosis [22].

Cyber-physical systems can be found in several different areas of expertise, such as aerospace engineering, automotive industry, healthcare, manufacturing, and many more. Due to the nature of these fields, human interaction is often limited and serves the role of observer or goal setter. Therefore, we expect those systems to be fully autonomous, secure, and fault-free. As faults will always be present, if compensating or repair actions can be performed without the need for human interaction; time, space and money saving is possible.
Consider for example offshore wind turbines used for generating electricity.

Reliance on them as a source of clean electricity will continue to grow, and the number of turbines will surely increase. If a fault occurs on one offshore turbine, manual repair actions are expected to be quite costly, both in terms of money and time, due to their location and expertise needed to perform repairs. However, if a wind turbine can automatically repair itself once a fault is known manual repair action is not needed. If the turbine however cannot perform necessary repair actions on its own, compensating actions may be used to postpone manual repair or to keep the turbine working at a lower capacity or even to gracefully shut it down to prevent further damage to the turbine. The application of abductive diagnostic reasoning for wind power plants was shown in [7].

The model used for diagnostic reasoning can be developed in several ways. Different levels of abstraction can be used to model a system, as well as different reasoning paradigms (qualitative or quantitative). A wide variety of options, with no choice being the correct one as well as trade-offs between scalability and diagnostic preciseness have to be taken into consideration when choosing abstraction level and description formats. Exploring different abstraction levels is one of CatIO's features, as once a model has been developed, it can be tested with a wide variety of simulations to see which behaviors it can distinguish.
Presented particularities and challenges resulted in the concept and implementation of CatIO. Therefore, the main purpose of this thesis is to ease the creation and validation of models used for diagnostic reasoning and hopes to bridge a gap between the benefits and costs associated with model-based diagnosis in real-world development.

The thesis is partially based on [26], where the author presents a way in which model-based diagnosis can be used in self-adaptive and autonomous systems. CatIO was designed in a way in which enables seamless implementation and integration of all concepts presented in [26].
The architectural concepts of CatIO were recognized and well received by peers as there is currently no framework, according to our knowledge, which integrates model-based diagnosis with repair, especially in the scope of cyber-physical systems. The outline of CatIO's architecture was accepted at the 25th International Symposium on Methodologies for Intelligent Systems conference [4].

1.2. Objectives and Scope

The goal of the thesis is to develop a framework that seamlessly integrates simulations of the cyber-physical systems with the model-based diagnosis. As previously stated, the development of the model suitable for diagnostic purposes is a cumbersome task and often a stepping stone to further integration of MBD in the project structure/system life cycle. To overcome this, CatIO provides two modeling environments, for both consistency-based and abductive diagnosis (Sections 2.5 and 2.6). Logic-based modeling grammars are presented (Section 3.2.2), as well as a graphical user interface (Section 3.4) in which the user can develop diagnostic models aided by some functionalities which ease the creation, as well as testing of said model.

Detailed simulations of cyber-physical systems can be developed in the Modelica [6] modeling language. CPS in Modelica can be modeled with a combination of logic, arithmetic, programming constructs, and differential equations. While the development of the simulation model in Modelica modeling language is not the focus point of the thesis, we will shortly explain how a certain type of model is more versatile and can be used by the framework to automatically or manually generate different simulation scenarios. Those models and corresponding simulation scenarios are then interacted with, and the practical feasibility of a previously developed diagnosis model can be explored. Likewise, the quality of the controller which handles repair/compensating actions can be explored on said fault injected simulations.
Finally, special attention is given to the automatic generation of an abductive model, as well as to challenges which a designer faces while implementing needed interfaces. Several models will be automatically generated, and problems that arose throughout automatic generation procedure noted.

Examples shown in the thesis are small and serve as a proof of concept, but given sufficient development time, designers could employ CatIO functionalities on real-life medium to large-sized projects.

1.3. Structure

In this chapter, we have presented the motivations and goals of the thesis. In the next section running example is defined. It will be used to demonstrate concepts throughout the thesis. In the Chapter 2, we will define preliminaries, with basic definitions and explanations of the concepts used and implemented in the framework.
In Chapter 3, we will document the framework's front-end, trough definitions, and examples, as well as describe the underlying design decision which is found in the back-end of the framework.
Chapter 4 introduces several systems, corresponding models, their diagnostic capabilities, compares different diagnosis approaches as well as evaluates automatically generated models.
Finally, in Chapter 5 we summarize the framework and give an outline for future work, as well as possibilities of integration of the framework in product/system development.

1.4. Running Example

A simple example of a cyber-physical system will be used to demonstrate some concepts described in preliminaries and framework documentation. For this purpose, we will use a mobile robot with a differential drive which is used to move from one point in a plain to another. The differential drive robot is shown in Figure 1.1.

A differential drive robot is characterized by having 2 wheels with independent speeds. Depending on the speed of each wheel the robot can either move in a straight line, make a curve of varying degrees, or rotate in place.

As seen in Figure 1.1, the position of the robot in the coordinate system is denoted by (X_R, Y_R), and it is heading in the direction of θ. Distance between wheels is denoted by d. The speed of the right and left wheel is determined by input voltage, V_R, and V_L respectively, cause the rotation of the robot around ICC (instantaneous center of curvature) with ω being

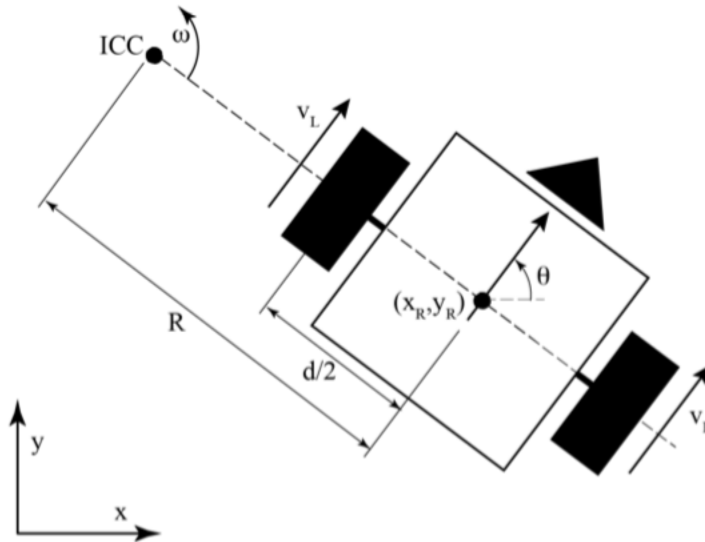

Figure 1.1.: A mobile robot with differential drive

rotational speed. Finally, R denotes the distance from the center of the robot to the ICC.

Differential equations that describe the robot's movement can be found in the Modelica model in Section 4.1.1. With respect to the speed of each wheel, the movement of the robot can be described as

- *speed(leftWheel) == speed(rightWheel)* $\rightarrow$ *direction(straight)*
- *speed(leftWheel)* $>$ *speed(rightWheel)* $\rightarrow$ *direction(right)*
- *speed(leftWheel)* $<$ *speed(rightWheel)* $\rightarrow$ *direction(left)*
- *speed(leftWheel, rightWheel)* $== 0 \rightarrow$ *direction(stop).*

Let us also introduce some faults, which will be used in this example throughout the thesis. In fault free behavior, wheels spin with the speed determined by voltage inputs V_R and V_L. However, let us assume that a wheel can spin *faster* or *slower* than expected. When the wheel spins faster or slower than expected, the heading angle of the robot will be compromised. These faults occur in the robot itself and consequently will result in undesired behavior.

2. From Logic to Diagnosis

The master thesis should be self-contained, so we present and explain underlying concepts that provide the reader with an understanding of the inner-working of CatIO. Detailed definitions of model-based diagnosis, different modeling principles, and their advantages and disadvantages, the formal definition of simulation. In this chapter, our goal is to show how concepts found in mathematical logic can be used to derive a diagnosis. Presented concepts are implemented and design decisions are shown in Section 3.

Throughout the thesis, we will deal with two different kinds of models: diagnosis models and simulation[1] models. Diagnosis models are used for diagnosis, and simulation models are used for simulation. If just the term model is used, we refer to the diagnosis model.

The following quote from statistician George Box refers primarily to statistical models, but it is frequently quoted in various domains of science to state the fact that we can never truly model complex real-world systems, but nonetheless we can create models that can be useful, in our case to derive a diagnosis.

> *All models are wrong, but some are useful.*
> *– George Box*

[1] In literature one can find the term "system model" often used for simulation models, but we will use the term simulation model to make a more clear distinction.

2.1. Logic

Logic is a product of the human reason with which we formulate unambiguous concepts that can be compared and related to each other. Starting from the small set of basic concepts called axioms, which are by definition not provable and assumed to be true we can form more complex concepts and ideas by applying simple rules.

Being a subset of philosophy and mathematics makes logic one of the oldest branches of thought and we will show how by using a small subset of logic we can come up with meaningful models and diagnosis.

Terms of predicate logic [18] used throughout the thesis are defined as:

- $true$ and $false$
- predicate is a Boolean valued function *P(X)* $\rightarrow$ *true, false*
- P is a sentence if p is a predicate
- $\neg P$ is a sentence which is negation of sentence P
- $(P \wedge Q)$ is a sentence where *and* is denoted by $\wedge$
- $(P \vee Q)$ is a sentence where *or* is denoted by $\vee$ denoted or operator
- $(P \rightarrow Q)$ is a sentence where *implies* is denoted by $\rightarrow$
- $(P \leftrightarrow Q)$ is a sentence where and *biconditional* denoted by $\leftrightarrow$

While constants $true$ and $false$, as well as basic operators as negation, conjunction (*and*), and disjunction (*or*), are inherently clear to us, in the next section we will explain how we can use implication and biconditional to come up with models.

2.2. From Logic to Model

The diagnostic model is a logical representation of some real-world or abstract systems. When describing an abstract system that can be fully expressed with logic there is no loss of information, but when describing some complex real-world system there are often too many factors to consider and we have to use abstraction.

Let us consider the example of AND gate in a logical circuit.

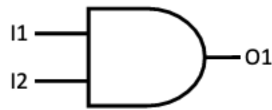

Figure 2.1.: And gate with inputs I1 and I2 and output O1

Obviously, it can be represented by logic without the loss of any information. We can model this gate either by defining all input combinations and corresponding output in a truth table or we can define a function which states how inputs correlate to an output.

Truth table defined as

$I1$	$I2$	$I1 \land I2$
T	T	T
T	F	F
F	T	F
F	F	F

can be expressed with following logical expression.

$$(I1 \land I2 \rightarrow O1) \lor (\neg I1 \land I2 \rightarrow \neg O1) \lor \\ (I1 \land \neg I2 \rightarrow \neg O1) \lor (\neg I1 \land \neg I2 \rightarrow \neg O1)$$

In this model, the implication is used to state that if a combination of inputs evaluates to true then O1 will also be true, as the only way for a sentence to be true if the left-hand side of implication is true is for a right-hand side to also be true.

Alternatively, we can use a biconditional operator. It implicitly states that output $O1$ has the same value as $I1 \land I2$.

$$I1 \land I2 \leftrightarrow O1$$

However, the real world is not comprised of systems that can be described with logic without losing some information. Not all possible inputs and external forces that may occur in cyber-physical systems can be expressed without abstraction.

Abstraction is a process of simplifying a model by removing details. There is no simple algorithm that can determine the level of abstraction which would satisfy the designer's needs and therefore several levels of abstraction may be considered and compared. Often time it is not clear which details of the system can be left out of the model without losing too much information which would otherwise improve the outcome of diagnostic reasoning.

2.3. From Model to Diagnosis

Model-based diagnosis[2] (MBD) is a subfield of artificial intelligence that encompasses different methods of finding causes of some unexpected or faulty behavior. As the name suggests, MBD is based on a model, which is a formalized description of the system under consideration.

A system can be formalized as a pair (*SD*,*H*) where the system description *SD* is a set of first-order/propositional logic or any other logic sentences while *H* is a finite set of health state variables. Observations of the system are a finite set of logical sentences. Observations describe outputs of the system or symptoms/behavior that can be observed. We shall write *(SD, COMPONENTS, OBS)* for a system *(SD, H)* alongside it's observation *OBS*. Formally, diagnosis problem and a solution to a diagnosis can be described as follows:

Definition 1 *A diagnosis problem can be described as a tuple* (SD, H, OBS)*, where the tuple* (SD, H) *describes the system to be diagnosed and OBS is a set of observations concerning its behavior.*

[2]from Greek *diagnOsis* - The art or act of identifying a disease from its signs and symptoms

A solution Δ to a diagnosis problem is a set of components c_i which, when assumed that these c_i behave abnormally (s.t. $h_{c_i} \in \Delta$), explains the conflicts between expected and observed behavior [9, 19].

Definition 2 *$\Delta \subseteq H$ is a diagnosis for a diagnosis problem (SD, H, OBS) if and only if $SD \cup OBS \cup \{h_i | h_i \in H \setminus \Delta\}$ is consistent (satisfiable) and there exists no $\Delta\prime \subset \Delta$ that is also a diagnosis.*

Suppose that we have a trivial system that was modeled in the previous section, comprising of only one *and* gate. Based on inputs of the system we can use our model to derive expected output.

To use this model for diagnostic purposes, we have to introduce the concepts of health state variables. A component's health state encodes the assumption of whether this component works correctly or not. The model with heath states can be expressed as

$$\neg Ab(AndGate) \rightarrow I1 \wedge I2 \leftrightarrow O1.$$

$\neg Ab(AndGate)$ is a health state predicate which states that component $AndGate$ is not behaving abnormally. As we can see from this model, we assumed that the *and* gate is not behaving abnormally and we have defined expected behavior in that case.

If we observe a discrepancy between observations of the physical system and expected behavior defined by the model, we are interested in components or sets of components whose simultaneous malfunction explains the issue. In this scenario where a discrepancy is detected diagnosis is performed. This procedure can be seen in Figure 2.2. For this system with one *and* gate, if expected and actual observations do not match, the only possible diagnosis would be $Ab(AndGate)$. Systems usually have multiple components, so algorithms like RC-Tree or ATMS are used to compute sets of components that may have caused such unexpected behavior.

The result of a diagnostic reasoning process is a set of sets of components. Each set of components in explains the discrepancy in the desired and observed behavior. For some systems it is possible to get one diagnosis,

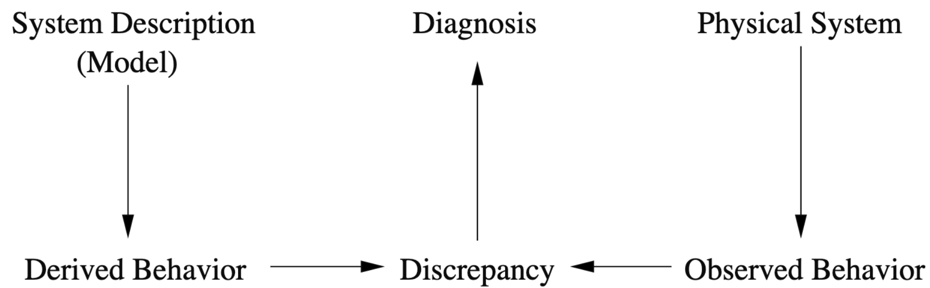

Figure 2.2.: Model based diagnosis

that being a one set of components which when behaving abnormally is explaining faulty behavior. Is there is a single diagnosis, this is ideal since then there is no uncertainty about the component(s) which had caused the error. When the diagnosis algorithm returns multiple diagnoses, we do not initially know which one describes the actual cause of the discrepancy.

Diagnosis can be seen as an explanation for given observations, in the case when observations do not conform to system description. We can obtain diagnosis either via conflicts, as described in Section 2.5 or via abductive reasoning described in Section 2.6.

Observations put a constraint on the quality of the model. If observations are too vague, they will result in a model that will hardly lead to the computation of a single diagnosis. However, to obtain better observations sometimes system redesign is needed or the addition of additional sensors/reading devices.

Suppose the example of the human body, and we want to use diagnostic reasoning to explain why the person is feeling unwell. If our only observation is body temperature, which may be higher than expected with varying degrees of severity, our assumptions about the cause of the sickness are very vague. Then the physician can apply some heuristic, probabilistic reasoning or his experience to narrow the possible diagnosis, but it is still unreliable. However, if a person performs the blood check and a separate check of the lungs and liver, the physician will have much more observations which he can use to rule out some diagnosis or even to obtain the single diagnosis.

We can see how the quality and quantity of observations can affect the quality of diagnostic reasoning. From only 2 observations (high temperature and feeling unwell) we can not precisely determine the root cause of said symptoms, but with more and precise observations (CTR, blood work, high temperature...) physician can derive more precise diagnosis.
While diagnostic reasoning, specifically abductive reasoning can be used in medicine [10], we will focus on technical systems.

2.4. Weak and Strong Fault Models

Before proceeding to diagnosis procedures, we have to discuss the difference between *weak fault models* and *strong fault models*. In weak fault models, system description describes the expected behavior of the system assuming that all components are working as expected. Meaning if the health state of a component suggests that it is faulty we do not impose any restrictions on this component's behavior.
On the other hand, strong fault models describe the expected behavior of the system as well as different fault modes for each component. Each fault mode describes how a component behaves if we assume that it is in that state like a *stuck at zero* fault for an logic *and* gate.

Component's behavior for weak fault mode description, with `AB(comp)` being unary predicate which is interpreted as *comp* is behaving abnormally, is as follows

$$
\begin{gathered}
\ldots \\
\neg AB(comp) \rightarrow ExptectedBehavior \\
\ldots
\end{gathered}
$$

whereas strong fault models are described as

$$\begin{gathered} \dots \\ NormalBehavior(comp) \rightarrow ExptectedBehavior \\ FaultMode_1(comp) \rightarrow FaultyBehavior_1 \\ FaultMode_2(comp) \rightarrow FaultyBehavior_2 \\ \dots \end{gathered}$$

where $FaultMode_i(comp)$ indicates specific fault mode of a certain component.

2.5. Consistency Based Diagnosis

Consistency based diagnosis (CBD) is based on the principle of consistency. In logic, a set of statement is consistent if all of the statements can be true at the same time. General idea of CBD is to describe the model of a system in some logical representation and for a given observations check if union of them and a model is still consistent under the assumption that all components work correctly [9].

If model and observations are not consistent under this assumption, then is entailed that some parts of the model have to be changed, in order to be able to explain these observations.
In order to restore consistency, health state variables of the model are set as abnormal in hopes of making the model and observations consistent. If some (or more) health state variables are set to abnormal and model and observations become consistent with the altered model we can conclude that components associated with said health state variable was the cause of inconsistency/discrepancy in observations.

Naturally, more than one component may be faulty at any time and determining which combination of components makes the model inconsistent with observations is NP-complete problem. Naive algorithm would be checking all possible combinations of components for consistency with observations but due to the exponential size of all possible components combination that approach is not used. However, with computation of conflicts (assumption about which components can be faulty) by extracting minimal

unsatisfiable subset and with diagnosis algorithm like HS-DAG or RC-Tree computational cost of diagnosis can be reduced.
CBD is usually associated with weak faulty models, but it can also be used with strong fault models [13].
Following definitions explain main concepts of CBD in more detail.

Definition 3 *A conflict C for (SD, H, OBS) is a subset of H such that $SD \cup OBS \cup \{h_i | h_i \in C\}$ is inconsistent (unsatisfiable). If no proper subset C' of C is a conflict, then C is a subset-minimal conflict.*

To understand how conflicts are computed, and consequently design decision for representation of consistency based model in conjunctive normal form, in Section 2.5.1 we briefly explain how conflicts are computed. Formally, Reiter [19] defines hitting sets as a solution to a diagnosis problem as follows:

Definition 4 *A hitting set Δ for a set CS of conflicts is a subset of H such that $\Delta \cap C_i \neq \emptyset$ for all $C_i \in CS$. Δ is a minimal hitting set if and only if no proper subset of Δ is also a hitting set.*

Definition 5 *A diagnosis Δ for (SD, H, OBS) is a minimal hitting set for the set of all conflicts for this diagnosis problem. Since Δ will hit also all non-minimal conflicts if it hits the minimal ones, it suffices to focus on the minimal conflicts only when computing the diagnoses.*

We have chosen RC-Tree as an algorithm for computing diagnosis. With RC-Tree conflicts can be computed *on-the-fly* and they can be "known" in advance. In the following sections, we will explain the inner workings of RC-Tree and how it can be used to compute diagnosis, as well as diagnosis up to cardinality k.

2.5.1. Unsatisfiable Subset Extraction

Conflicts are the main building block of consistency-based diagnosis. A brief explanation of how we derive conflicts will help us understand other design decisions behind consistency based models and their logical representation.

Suppose that we have a Boolean propositional formula in conjunctive normal form and that it is unsatisfiable. An unsatisfiable subset (or core) is a subset of clauses whose conjunction is still unsatisfied. Furthermore, such an unsatisfiable subset is minimal, if every proper subset of it is satisfactory.

The simplest approach of computing an unsatisfiable subset of an unsatisfiable formula would be dropping one literal at the time and checking if the formula is then still unsatisfiable. If it is, we continue the procedure with this newly created formula. We do this for all atoms and the resulting formula is a minimal unsatisfiable subset. This is the most trivial approach, and we would need n satisfiability queries if there are n literals in the formula. One can approve this approach by using a binary search concept, initially removing half literals at the time; or by using some other method of finding an unsatisfiable subset.

SAT solvers can produce a resolution graph that proves the unsatisfiability of the original problem which can be analyzed to produce a smaller unsatisfiable core. We will treat unsatisfiable subset extraction as a black box, as it is not in the scope of the thesis. Tool *PicoMUS* is used to extract a minimal unsatisfiable subset. Result obtained from the PicoMUS is processed and only variables that correspond to the health state variables are considered in further computations.

There is also the route to compute the diagnosis directly in the solver. The interested reader may take a look at [11] for an outline of the SAT-based encoding that can be used to compute diagnosis directly in an SAT solver. A comparison of such an approach to other approaches can be seen in [12].

2.5.2. RC-Tree

RC-Tree [14] is a variant of the HS-DAG [8] algorithm used for computation of minimal hitting sets. As seen in Section 2.5, Reiter showed that the computation of minimal hitting sets correspond to a solution of the diagnosis problem, therefore by using RC-Tree and user-provided model we can compute diagnosis of a said model for given observations.
RC-Tree was chosen over more wide-spread HS-DAG, as it in the vast majority of cases perform better due to the avoidance of redundant branches.

Like for HS-DAG, the basic concept for RC-Tree is to construct a tree (a directed acrylic graph for HS-DAG) such that the leaves marked with a "checkmark" represent the minimal hitting sets. Union of leaves forms a subset of minimal diagnosis.

Each node in a three/DAG is characterized with $h(n)$, that being path from the root up to a node. The path consists of edges which hold a health state variable. Special for RC-Tree is that we do not construct any $h(n)$ more than once, so that any diagnostic candidate $h(n)$ is investigated in one branch only. To that end, each note has a filter $\Theta(n)$. Each filter contains a health state variable that will not be further processed in children's branches. By doing so the creation of redundant notes with the same path is avoided. Note that the path is seen as a set where the ordering of the values does not matter ($\{1,2,3\} = \{3,2,1\}$) as the diagnosis is a set (not ordered) of the component.
The real run-time benefit which RC-Tree has over HS-Dag is a smaller final tree, which entails fewer calls to SAT solver which is the most resource-intensive part of CBD.

Suppose that computed conflicts are sets $\{\{1,2\},\{1,4\},\{1,5,6\}\}$. For those conflicts subset minimal hitting sets would be $\{\{1\},\{2,4,5\},\{2,4,6\}\}$. Notice that $\{1\}$ is in all three conflicts, therefore it is a subset minimal hitting set. Hitting set $\{1,4\}$ is not subset minimal, as it is super set of $\{1\}$.

RC-Tree starts with the root node with an empty filter for which a label is computed on-the-fly. A label is a numerical representation of components that can be faulty or of a certain fault state.
Then for every element of the label new node is created. Nodes are created

and processed in the breadth-first order. Once a node is created it's filter is set to a union of its parent filter and all current sibling (children from the parent).
For each node closing check is performed, that is to check if the node is a superset of some minimal hitting set, as well as pruning check and pruning procedure.
Pruning[3] is a procedure where certain branches of the tree are removed as they are shown to be redundant, ie. they will always result in closed nodes as they will never be subset minimal. This occurs if a node has been labeled by a label which is a superset of some other label. In principle, while pruning the label is then replaced by the subset in the whole tree and the tree is updated accordingly (see [14] for more details). An empirical evaluation has shown that pruning provides a substantial decrease in run time, prematurely on larger examples, whereas on smaller examples with smaller tree sizes computations performed while pruning can increase the run time [15].

The whole RC-Tree algorithm is shown in Algorithm 1.

As the RC-Tree's authors suggested computation of the labels in CatIO is done *on-the-fly*. To achieve that, the model is changed for every node, namely health state variables corresponding to ones found in the path from the root are set to abnormal. Then with PicoMUS check if such a model is satisfiable with respect to the observations. If it is satisfiable, it is entailed that before mentioned components are not behaving as expected and form a diagnosis. In the case of unsatisfiability, PicoMUS will return health states which then form the label of the node in question.

[3]To prune - cut branches from the tree

2.6. Abductive Diagnosis

As the name suggests, abductive reasoning is the main concept behind abductive diagnosis [3]. In logic abduction, alongside induction and deduction, is a type of logical inference. Abductive reasoning aims to find an explanation for a given observation with respect to the knowledge it has. Therefore the main challenge in abductive reasoning is to come up with appropriate knowledge base *KB* which can explain most, if not all observations.
In MBD, such a knowledge base describes dependencies between faults and their effects considering the system's observable behavior. The knowledge base then allows us to abductively reason backward from experienced symptoms to possible root causes [5]. Due to the nature of reasoning, such models/KB are less detailed than consistency based variants, as they do not specifically state normal and abnormal behavior of components, but more abstractly describe behavior if the form of *cause* $\rightarrow$ *effect* statements. Following a formalized definition will help us understand abductive diagnosis in more detail.

Definition 6 *The knowledge base is a tuple* (A, Hyp, Th) *where* A *denotes a subset of propositional variables* $PROPS$*, hypotheses(*Hyp*) are a subset of* A*, and theory* Th *is a set of Horn clauses over* A*.*

Hypothesis corresponds directly to causes and throughout the thesis, we will use the terms interchangeably.

Definition 7 *Propositional horn clause abduction problem* $PHCAP$ *is a tuple* (A, Hyp, Th, Obs)*, where* (A, Hyp, Th) *form a knowledge base and* OBS *are observations such that* $OBS \subseteq A$*. Solution to* $PHCAP$ *is a set* $\Delta \subseteq Hyp$ *if and only if* $\Delta \cup Th \models Obs$ *and* $\Delta \cup Th \not\models \perp$*.*

A solution to $PHCAP$ is the abductive diagnosis.
Δ is an explanation for a given observations. Δ is subset minimal if there is no $\Delta\prime \subset \Delta$.

We used Assumption based truth maintenance($ATMS$) as a reasoning system. With ATMS we can derive explanations Δ for the abductive diagnosis.

2.6.1. Assumption Based Truth Maintenance System

Assumption Based Truth Maintenance System(ATMS) [20] is a problem solver module which we use for computing abductive diagnosis. Its features include eliminating inconsistencies in the knowledge base, computing explanations for conclusions, and updating knowledge base as well as justifications.

ATMS' main task is to ensure consistency, which is done by changing the labels of the nodes.

For clearer understanding of the algorithm, following definitions are useful.

- **Node** - corresponds to a problem-solver datum
- **Justification** - horn clauses which describe how nodes are derived from other nodes
- **Assumption** - a special node
- **Environment** - a set of assumptions
- **Characterizing environment** - minimal consistent environment from which a context can be derived
- **Context** - formed by a consistent environment and all nodes derived from it.

The environment contains a node n if and only if n can be derived/entailed from the environment and current theory. Furthermore, the environment is inconsistent if the false node can be derived from it. Every node has a label, which corresponds to a set of sets of assumptions (consistent environment) from which the node can be derived and from which the contradicting node cannot be derived.
Each label fulfills the following properties: consistency, soundness, completeness, and minimality. The Task of ATMS is to compute the node labels.

Each node has a label, i.e., a set of sets of assumptions from which the node can be inferred and from which the contradicting node cannot be inferred. The latter requirement causes an ATMS algorithm to remove elements from the label that also lead to the NoGood. Hence, an abductive explanation for a single proposition is an element of the label of the corresponding node. Because of the ATMS these elements provide a consistent explanation and

fulfill the definition of abductive diagnosis. The only thing that remains now is the extension to the case where we have a set of observations to be explained. This extension can be easily done by adding a rule where the left side is the conjunction of all observations and the right side is a new proposition "explain" not used in the KB. Hence, the label of the "explain" proposition provides all abductive diagnoses for the given PHCAP.

2.7. Mixed level covering arrays

Mixed level covering arrays (MCA) are used in combinatorial testing to deal with the exponential growth of possible combinations of all variables and their values.

Definition 8 *A mixed-level covering array is defined as* $MCA(I, (A_1, \ldots, A_k), s)$ *of strength* s *for* $k = |I|$ *variables with their individual finite alphabets* A_i *is a two-dimensional* $k \times m$ *array such that for any* $I' \subseteq I$ *such that* $|I'| = s$ *we have every combination in the cross product of the individual alphabets of the variables in* I' *appears in at least one of the* m *rows.*

2.8. Models and Simulation

Throughout the thesis, we described the model used for diagnosis purposes. Alongside this model, we will also use the model of a system that will simulate the physical system to the best of its abilities. Limitations that occur while modeling (cyber) physical systems are inherent in the task of modeling. We can not model every possible smallest physical interaction or the force which can be exerted on the model due to cost or other constraints.

Nonetheless, those limitations do not prevent simulation models to be used throughout the research and industry. Once a system model has been developed, it can be used for various forms of testing and verification instead of the physical model.

For the presented work, we used Modelica modeling language to develop simulation models of the system. Observations necessary for diagnostic reasoning are obtained from simulations performed with said models. Without simulations, the user would have to provide realistic observations manually or by using some other technique.
Those models have to enable fault injection. Fault injection [24] is a process of injecting a fault in the model while continuing to simulate its behavior. Fault injection creates an observable difference (in the simulation modeling environment) in between fault-free and fault injected simulation.

A simulation model is used to describe a system, using logic, mathematics, and other concepts found in Modelica modeling language, while diagnosis models are simply logical formulas that describe the system on an abstract level to be able to find, hopefully, a single diagnosis if some fault occurs.

More formally, the simulation model and simulation are described in [16] as follows.

Definition 9 *Simulation model M is a tuple $(COMP, MODES, \mu, I, O, P)$, where $COMP$ is a finite set of components, $MODES$ is a finite set of modes with at least one element, that being correct mode ok. Mapping function $\mu : COMP \rightarrow MODES$ maps components to one of their modes. I is a set of variables considered as input, while O is a set of output variables. Finally, Modelica Model P with fault injection capabilities allows setting a mode $m \in \mu(c) | \forall c \in COMP$.*

The simulation model in itself cannot be used for simulation by Modelica, therefore we need to introduce the concept of simulation and mode assignment.

Definition 10 *Mode assignment is a function which assigns a mode m for each component $c \in COMP$, for all time points. Time points are a finite set denoted by $TIME$. Mode assignment is then a function $COMP \times TIME \rightarrow MODES$. If a set of inputs I is defined all inputs need to be defined over time. Inputs over time can be denoted with a function $I \times TIME \rightarrow val(I)$ where $val(I)$ is a function mapping input variable to a set of all possible values which that input can have.*

Simulation can be seen as a function which computes all the variable values of P over time, with respect to system inputs, mode assignment, and simulation run time.

Any other modeling language and environments such as Simulink or MATLAB can easily be added to the framework, as they only need to support the extraction of the model/simulation to the functional mock-up interface.

Algorithm 1 Computing diagnoses using RC-Tree

Require: precomputed conflicts set(CS)or ability to compute conflicts on the fly

1: Let D be a growing node and edge-labeled tree with some initial and unlabeled root node n_0. Unlabeled nodes are processed in breath first order. Each node n is defined by $h(n)$, which represents path(set of edge labels) from root node n_0 to node n. $h(n_0)$ is an empty set. Each node has it's accompanying filter Θ, with filter of root node being empty set.
2: **procedure** RCTREE
3:     $MHS \leftarrow \{\}$ ▷ Minimal hitting sets
4:     $nodesToProcess \leftarrow \{rootNode\}$
5:     **while** $nodesToProcess \neq \emptyset$ **do** ▷ Check for closing
6:         $nodeInProcess \leftarrow nodesToProcess.pop$
7:         **for** $hs \in MHS$ **do**
8:             **if** $hs \subset h(nodesToProcess)$ **then**
9:                 $mark\ node\ with\ \times$
10:                 $continue$
11:             **end if**
12:         **end for**
13:         $label \leftarrow computeLabel(nodeInProcess)$
14:         **if** $label = \emptyset$ **then**
15:             $mark\ node\ with\ \checkmark$
16:             $MHS \leftarrow MHS \cup h(nodeInProcess)$ ▷ $h(node)$ is a diagnosis
17:         **end if**
18:         $pruneTree(label)$
19:         **for** $edge \in label$ **do**
20:             **if** $edge \in \Theta(nodeInProcess)$ **then**
21:                 continue
22:             **end if**
23:             $childNode \leftarrow h(nodeInProcess) \cup edge$
24:             $\Theta(childNode) \leftarrow \Theta(nodeInProcess) \cup nodeInProcess.children$
25:             $nodesToProcess \leftarrow nodesToProcess \cup childNode$
26:         **end for**
27:     **end while**
28: **end procedure**

3. Design and Implementation of a Diagnosis Framework

In this chapter design decisions and implementation will be presented and explained in detail. Some concepts found in the preliminaries chapter will be expanded. Alongside the design of the framework, simple code snippets will explain the main points of interaction between the user and the framework.

A high-level representation of CatIO's architecture can be seen in Figure 3.1. The remainder of the chapter will be organized with respect to this figure. Firstly, the whole front end will be explained in detail with provided examples, as in the expected use of the framework user interacts only through the front end. Additionally, in Section 3.2.1 2 ways of modeling in Modelica will be explained and their benefits and drawbacks with respect to the framework. Additionally, CatIO's graphical user interface will be presented.

Throughout this chapter, we will use the differential drive robot introduced in Section 1.4 for demonstration purposes.

3.1. Workflow

In this section, we will present the expected workflow of the CatIO.
In Figure 3.2, we can see CatIO's inputs and outputs. The inputs include a model(s) used for diagnostic purposes, implementation of needed interfaces, as well as a Modelica system model exported as Functional Mock-up Interface (*FMI*).

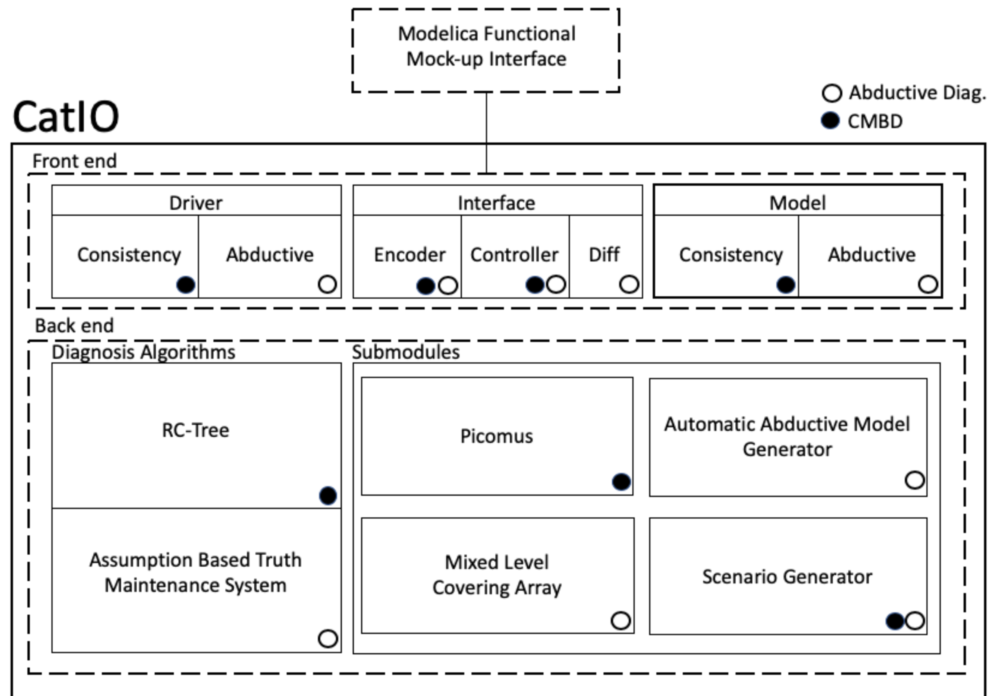

Figure 3.1.: CatIO's arhitecture

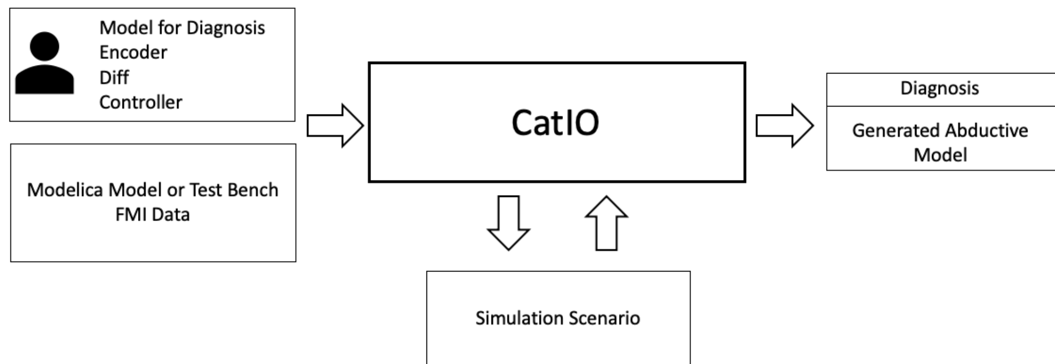

Figure 3.2.: CatIO's inputs and outputs

CatIO can read and write data to the simulation (model). From the simula-

tion, it obtains values that are used as observations of a system. CatIO can inject faults into the simulation (used for dynamic generation of scenarios) or write values that are used to perform repair actions.

FMI [2] is a standardized interface used in the development of complex simulations. With FMI one can run the simulation and interact with it using external programs, in the case of CatIO with the use of a *fmu-wrapper* library. Note that through this chapter we will be referring to a *.fmi* file of a system, which is a file containing all needed data for simulation of our running example.

CatIO's outputs are then the diagnosis of executed simulation or automatically generated abductive model. To assess the capabilities of repair actions CatIO can also provide plotted variable data of the simulation values (as shown in Section 4.1.2).

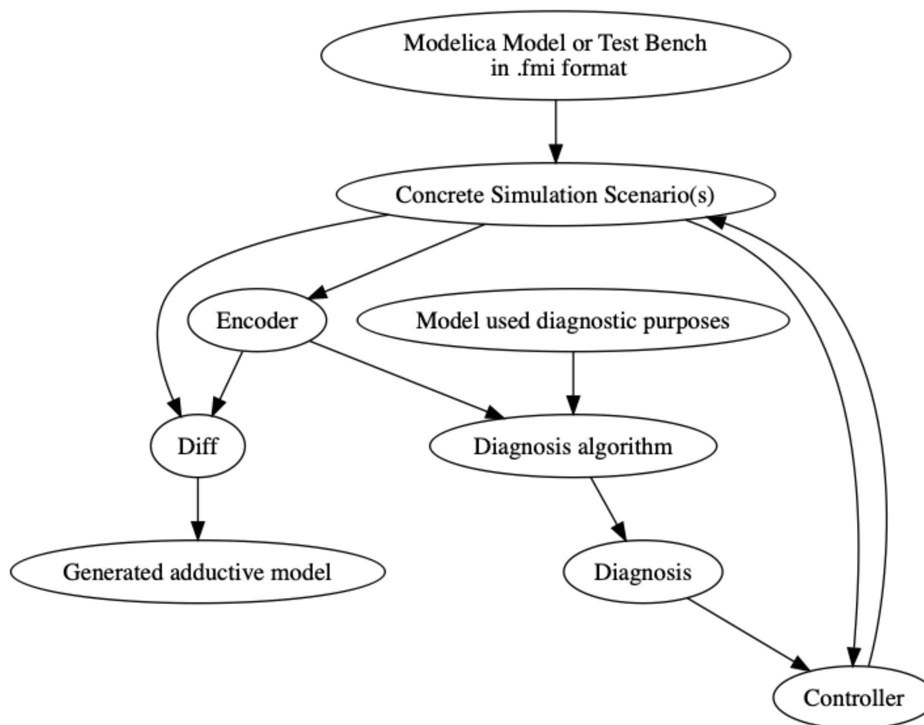

Figure 3.3.: CatIO's high level module interactions

In Section 3.3, we can see that in order to be able to derive diagnosis, a diagnosis algorithm is needed, as well as the model and encoded observations. `Encoder` maps concrete simulation values to observations to prepositions found in model.
Such encoded observations of multiple correct and fault injected simulations can be analyzed by `Diff` interface implementation and based on detected discrepancies abductive model can be generated. Furthermore, `Diff` interface can receive data directly from the simulation.
Based on the computed diagnosis, `Controller` can perform actions against a simulation by writing data to it. `Controller`'s decision making is aided by the ability to read data directly from the simulation.

3.2. Front end

3.2.1. Modelica Models

Modelica [6] system models are usually accompanied by a test bench which serves as an input provider for a simulation. Inputs are values that are assigned to the input variables, as well as fault modes that are used for fault injection. Test benches consist of a system under test (*SUT*) as well as input variables/mode assignment at defined time steps. The format of a standard test bench is of a format seen in Listing 3.1.

Listing 3.1: System with corresponding test bench

```
model Testbench
  SUT sys;
  equation
    if(time < t1) then
      ....  // First inputs
    elsif (time >= t1 and time < t2) then
      .... // Next inputs
    elsif
      ....
    else
      ....
    end if;
end Testbench;
```

Therefore, we propose *input oriented model*. In such a model, all variables of interest are set as inputs, and they can then be dynamically changed during simulation, which is not only needed for the dynamical generation of test scenarios, as the controller also needs to be able to interact with the simulation.

Such *input oriented model* is defined as

Listing 3.2: Input oriented model

```
model SUT
  input FaultTypeConnector modeAssigement_1;
  ...
```

```
   input FaultTypeConnector modeAssigmenent_n;
   input RealInput modelRealValue_1;
   ...
   BooleanInput modelBooleanValue_1;
   equation
     ...
end SUT;
```

The only prerequisite to simulating such a model is to provide values for all input parameters at the initialization of simulation. Otherwise, simulation can not be executed. How to generate simulation scenarios manually or via combination exploration of possible inputs is shown in Section 3.3.1.

Input oriented model as well as the standard model with accompanying test bench of differential drive robot can be found in Section 4.1.1.

Model Data Description

Once an FMI of a model or test bench has been provided to CatIO, the user needs to extract some values from the *.fmi* file. `ModelData` class contains all necessary description data of the *.fmi* file. Following member field describe the `ModelData` class.

- **Components to Read** These components will be read at every time step of the simulation and their values will be passed to *Encoder* implementation
- **Mode Assignment Variables** Variable names and values which are used for mode assignment (fault injection)
- **System Inputs** Inputs of the system, contain names and values which can be used in the automatic creation of test scenarios
- **System Parameters** Parameters of the system, contain names and values which can be used in the automatic creation of test scenarios
- **Plot Variables** Pair of strings, representing variable names found in .fmi, which are going to be plotted. The first variable corresponds to x, while the second corresponds to the y-axis
- **Controller** Controller which can be used to perform repair or compensating actions.

Mode assignment variables were explained in Section 2.8.
Components to Read are of a type `Component` which is defined in following listing. `Component` class contains string representing variable name of some component found in *.fmi* file, `TYPE` it's type, which can be either an {*STRING, BOOLEAN, DOUBLE, INTEGER, ENUM*} and finally `value` is a variable which stores a value read at every time step.

Listing 3.3: Component class

```
public class Component {
  String name;
  Type type;
  Object value;
}

// Signature of the constructors
public Component(String name, Type type);
public Component(String name, Type type, Object value);
```

System inputs are inputs that the system receives from the environment or some external controller, while system parameters are usually constant values of the inner state of system components. In our running example, system inputs are left and right wheel voltages, while the system parameter is the distance between wheels. Another example of system parameters is the resistance value of the resistor. While all resistors behave identically, depending on their parameter value they produce different outputs. Changing a resistors parameter value from 5Ω to 100Ω changes it's the effect on the system as a whole.

`ModelInput` is a data class describing all possible values which one of the three categories can take when automatically creating test scenarios.

Listing 3.4: ModelInput Class

```
public class ModelInput implements Serializable {
  String name;
  Type type;
  // All values which are going to be considered when
     creating mixed level coverring array
  List<Object> values;
}
```

In `ModelData` class, mode assignment variables, system inputs and system parameters are stored in list of `ModelInput` type, unlike components. Finally, the `ModelData` class is defined as follow:

Listing 3.5: ModelData class with public setters

```
public class ModelData {
  // Mandatory
  List<Component> componentsToRead;
  // Needed for custom simulation scenarios
  // and for mixed level coverring array generation
  List<ModelInput> modeAssigmentVars;
  List<ModelInput> systemInputs;
  List<ModelInput> systemParams;

  // Cannot populate with GUI, use setters
  Controller controller;
  Pair<Component, Component> plotPair;
  // Component plotOverTime;

  // Setters used for setting controller and plot variables
  // Can be left undefined
  public void setController(Controller cont);
  public void setPlotVariables(String xVar, String yVar);
}
```

To create `ModelData` of a $.fmi$ file, use of graphical user interface is recommended. Usage with example will be shown later. Note that the user may set two variable names in the plotting pair defined in `ModelData`, and after the experiment, values will be plotted. The first variable is shown on the x-axis, while the second is on the y-axis.

3.2.2. Diagnosis Models

Consistency Based Model

Consistency based models can be described using simple propositional logic. Logical operators found in this simple language are negation (!), conjunction (&), disjunction (|), implication ($\rightarrow$) and biconditional ($\leftrightarrow$).
Predicates $\$true$ and $\$false$ have true or false values respectively. Example use of $\$false$ predicate is to state impossibilities, in a clause if a form $(predicate_1 \wedge predicate_2 \wedge ...predicate_n) \rightarrow \$false$.
Literals/predicates starting with the capitalized letter are considered as health state variables and will be taken into consideration during unsatisfiable core extraction and thus diagnosis.

As explained in Section 2.5.1, to be able to extract a minimal unsatisfiable subset using SAT solvers, the model has to be in conjunctive normal form. Therefore the model described with this grammar is automatically converted to CNF. Every propositional formula can be converted to CNF [21] by using logical equivalence rules (De Morgans' Law, double negation elimination, and distributive law).
However, if a user wants to use in some way optimized/reduces CNF encoding, he can also create models by describing the CNF using comma-separated predicates, and dot-separated clauses.

Once a model is in CNF, every predicate is mapped to a unique integer. By doing so, we can express any consistency-based model in DIMCAS format. DIMCAS is a standard encoding of CNF used by the majority of SAT solvers. Once observations have been made, they to are mapped to integers and added to the model, and passed to the SAT solver to check for satisfiability.

Listing 3.6: Grammar for consistency based models.

```
Health State ::= [A-Z][A-Za-z0-9_]*
Variable ::= [a-z0-9_@][A-Za-z0-9_]*
True ::= '$true'
False ::= '$false'
Negation ::= '!'
Operator ::= '&' | '|' | '->' | '<->'
Literal ::= Variable | HealthState | True | False |
    Negation Literal
Formula ::= Literal | '(' Formula ')' | Formula Operator
    Formula
Clause ::= Literal (',' Literal)* '.'
ConsistencyModel ::= (Formula '.')+ | (Clause '.')+
```

Both weak and strong models, as shown in 2.4, can be described using this grammar.

An example of a logical circuit is ISCAS Benchmark Circuit c17, which is shown in Figure 3.4. It comprises 5 inputs labeled with $1, 2, 3, 6, 7$, two outputs $22, 23$ and 5 *nand* gates. Nand gate is defined as the negation of result of an and operator, or formally $NAND(I1, I2) = NOT\ AND(I1, I2)$

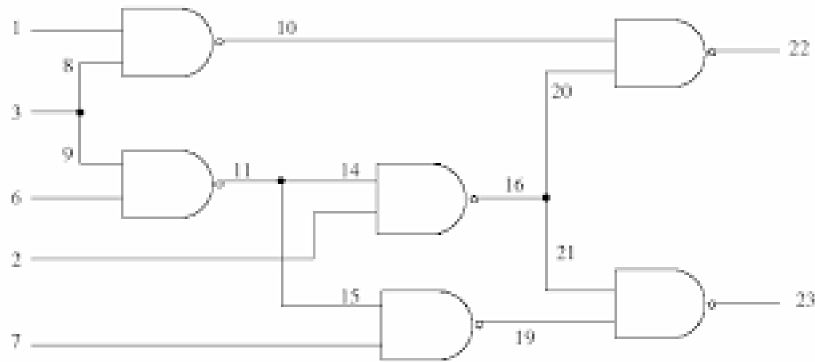

Figure 3.4.: ISCAS Benchmark Circuit c17

This system can be modeled as follows. *nand* literal represents *nand* gate. Note that *AbNand*{1..6} are predicates starting with capitalized letters and therefore can be the result of the diagnosis, as they represent health state variables.

Listing 3.7: Example model of ISCAS85-c17 circuit..

```
// logic behind the circuit/ connections of components
!AbNand1 -> (!( input1 & input3) -> output10).
!AbNand2 -> (!( input3 & input6) -> output11).
!AbNand3 -> (!( input2 & output11) -> output16).
!AbNand4 -> (!( output11 & input7) -> 19).
!AbNand5 -> (!( output10 & output16) -> output22).
!AbNand6 -> (!( output16 & output19) -> output23).
```

`CbModel` class is responsible for the creation of a consistency-based model that can be used by CatIO. The user only needs to pass the path to the file which contains the model described by before mentioned grammar to the `CbModel` constructor and the model will automatically be translated to CNF.

The usage of a graphical user interface is recommended for the development of the model. Following snippet shows how to get a diagnosis for manually provided observation without the use of GUI:

```
// Run diagnosis where model is CbModel object, and
List<String> obs = ... // observations
List<Integer> observationsToInt =
   model.observationToInt(obs);
RcTree rcTree = new RcTree(model, observationsToInt);

List<List<String>> diag = new ArrayList<>();

// for each minimal hitting set, map results to predicates
   used in model
for (List<Integer> mhs : rcTree.getDiagnosis())
  diag.add(model.diagnosisToComponentNames(mhs));
```

Abductive Model

Unlike consistency based models where detailed behavior of a system can be explained using propositional logic, in abductive modeling, we rely on implication relation, in a form of *cause* $\rightarrow$ *effect*. Following is the grammar used to define abductive models. Assumptions and hypotheses that are represented by a predicate have to start with a capitalized character. Additionally, the $false$ predicate is used to state contradictions or impossibilities.

Note that this grammar is also used when automatically creating abductive models.

Listing 3.8: Grammar for abductive models.

```
atom ::= id opt_args } | ϵ
opt_args ::= '(' args ')' | ϵ
args ::= term args_rest | ϵ
term ::= id opt_args
args_rest ::= ',' term args_rest | ϵ
rules ::= rule rules | epsilon
rule ::= atom rule_rest '.'
rule_rest ::= ':-' atom_list | atom_list_rest '->' atom
atom_list ::= atom atom_list_rest | epsilon
atom_list_rest ::= ',' atom atom_list_rest | ϵ
```

`AbModel` is a class analog to before mentioned class `CbModel`. User provides path to the file containing abductive model defined by corresponding grammar and ATMS will be constructed automatically.
`AbModel::tryToExplain(List<String> obs)` method is used to add observations and `AbModel::getDiagnosis()` method is used to get explanation.

```
...
List<String> obs = ... //observations
abductiveModel.tryToExplain(obs);
System.out.println(abductiveModel.getDiagnosis());
```

3.2.3. Interfaces

Encoder

`Encoder` is the most basic interface found in CatIO. Its implementation is required if the user wants to validate models' diagnostic capabilities with simulation scenarios. The encoder interface is needed as a bridge between raw simulation data/values and predicates found in the model. Models are created with the finite set of predicates while values found in the simulations are not necessarily in a finite domain.

Encoders method `encodeObservations` is in every time step of the simulation, and it's return value processed in one of the ways described in Section 3.2.4.

Values in simulations are of internal types found in Modelica modeling language, namely *Real*, *Integer*, *Boolean* and *String*. Observation *OBS* in both types of diagnosis are not from this domain, they are rather an interpretation of single or multiple values received from the simulation at any given time. Thus, the purpose of `Encoder` interface is to map values from before mentioned domains to literals/propositions found in the model.

Inputs to the `encodeObservation` method is a map connecting variable names (`ComponentsToRead` field of the `ModelData` class) found in the simulation with corresponding values which are in one of the before-mentioned types.

`Return` value of the function is therefore a set of strings, which represent propositions found in the model. Propositions that are false begin with "!".

In the following chapter, we will show a sample model of the differential drive robot as well as it's accompanying encoder.

```
public interface Encoder {
  /**
  * @param obs map containing names and values read from
     simulation
  * @return set of observations with respect to the model
  */
  Set<String> encodeObservation(Map<String, Object> obs);
}
```

Listing 3.9: Encoder interface.

Diff

`Diff` interface is used for the automatic generation of abductive models. In particular, the algorithm described in Section 3.3.1 employs a `Diff::encodeDiff` function to deduce the effects of injected faults. This function (also provided by the user) receives two `SimulationRunData` objects, which contain either values of the corresponding simulation or observations encoded with `Encoder` interface. `encodeDiff` identifies deviations between fault free and fault injected simulation and maps them to predicates which describe the discrepancy.

Listing 3.10: Diff with encoded values

```
public interface Diff {
  /**
  * @param correct SimulationRunData  values or predicates
     of simulation without fault injection
  * @param faulty SimulationRunData  values or predicates
     of simulation with fault injection
  * @return encoding of diff function with respect to the
     model
  */
  Set<String> encodeDiff(SimulationRunData correct,
     SimulationRunData faulty);
}
```

`SimulationRunData` is a helper class which contains all values or predicates (if `Encoder` was used). Users can then for every time step of the simulation

get either map containing names of variables found in $.fmi$ with corresponding value at that time step, or predicates which were returned by an encoder.

Listing 3.11: SimulationRunData class

```
public class SimulationRunData {
  private List<Map<String, Object>> valuesMap;
  private List<List<String>> predicatesMap;

  // returns values from step n
  public Map<String, Object> getValuesFromStep(int n);

  // returns encoded predicates from step n
  public List<String> getPredicatesFromStep(int n);
}
```

In principle, there are two concepts for implementing $diff$ function, namely qualitative and deviation models. Qualitative models map observations in some qualitative domain, whereas deviation model maps faulty observations with respect to the correct observations. Derivation of qualitative models is only possible from $diff$ function with direct read values, whereas deviation models can be derived from both $diff$ functions, with direct or previously encoded values.

Suppose that the left wheel is spinning faster than expected due to the slippery surface and that such deviation was noticed by $diff$ implementation. In deviation model, such observation would be encoded as

$$slippery(leftWheel) \rightarrow faster(leftWheel).$$

Qualitative model representation of this observation could be

$$slippery(leftWheel, 10) \rightarrow spinSpeed(leftWheel, 12)$$

where 10 was the initial and expected speed of the wheel, and 12 was the actual one.

Controller

`Controller` class is used to let the user evaluate the performance of compensating or repair actions with or without a diagnosis. `Controller` performs an action against a running simulation when a discrepancy is detected and diagnosis obtained.

The user has to implement a controller with respect to the *.fmi* file. As stated in Section 3.2.1, all values that can be changed during simulation have to be set as inputs. Interaction with inputs can then be performed by the controller. Compensating actions are actions that try to enable the correct behavior of the system in which some components do not behave as expected. Compensating actions can be used for safety-critical situations where it is important to reach a certain state before more complex repair actions can be taken and to avoid further degradation of the system. Repair actions can be seen as changing the state of the component from faulty to correct. In CatIO, repair action is then simply a mode assignment from faulty to correct, as we assume that the system has the capabilities for automatic replacements of components.

Listing 3.12: Controller interface

```
public interface Controller {
/**
  * @param fmiConnector fmiConnector used to write(and
     read) from current simulation
  * @param diagnosis single diagnosis or signal used to
     determine which action to take
  * @return remaining time steps of the action
  */
  int performAction(FmiConnector fmiConnector,
     List<String> diagnosis);
}
```

In the `Controller` interface, `performAction` method returns an integer, which represent remaining duration of an action. The duration of action is expressed in the simulation steps. Duration of simulation step is defined in `Driver` classes, as seen in Section 3.2.4. Idealistic repair actions are performed in a single step, therefore return value can be set to 0 as no more

steps are taken after the component has been repaired.
Compensating actions can however last several steps and the duration of the compensating action can even change depending on the system and how it reacts to certain actions. In the following chapter, we will demonstrate compensating actions for differential drive robot lasting several steps.
When possible, the logic of compensating actions should be implemented in the system model in Modelica, as in model all necessary variables/values are present. If a compensating action was implemented in Modelica, the user would simply trigger it in the `Controller`.

3.2.4. Drivers

`ConsistencyDriver` and `AbductiveDriver` classes are used to bind models, corresponding interfaces, and simulations together, as well as to select a certain type of diagnosis. Aside from the driver classes themselves, we will explain possible diagnosis options found in CatIO in this section, as they are selected by drivers.

Data that has to be passed to the drivers are a path to the .fmi file containing a simulation scenario or input-oriented model, a path to the file describing the model used for diagnosis, corresponding encoder as well as a simulation run time and step size.

Step size determines how often observations will be encoded. Step size defines a time interval in seconds after which values will be either read or written to the simulation. For example, if the step size is set to 4.3, after 4.3 seconds of the Modelica's simulation `ComponentToRead` variable values are going to be read and passed to an `Encoder`.
The total number of steps in the simulation run is the quotient of run time and step size. Bigger step size will make the simulation and diagnosis process faster but at the expanse of diagnosis quality, as some faults may not be noticed at all, noticed too late, or even masked by some other faults. Determining the right step size is done through trial and error, by making an educated guess or by matching its value with a sensory data frequency of the system.

In Listing 3.13, we can see the builder responsible for the creation of the `ConsistencyDriver` object with all necessary data provided.

```
ConsistencyDriver simpleDriver = ConsistencyDriver.builder()
    .pathToFmi("FMIs/ERobot.SubModel.InputSimpleRobot.fmu")
    .model(new
        CbModel("src/main/java/examples/extendedRobotModel.txt"))
    .encoder(new SimpleRobotEncoder())
    .modelData(modelData)
    .numberOfSteps(20)
    .simulationStepSize(1)
    .build();

// If using input-oriented model
// Simulation scenarios are importet from JSON
List<Scenario> extendedScenarios =
    Util.scenariosFromJson("extendedScenarios.json");
// Diagnosis is performed on scenario, with assumption that faults are
    intermittent
simpleDriver.runDiagnosis(ConsistencyType.INTERMITTENT,
    extendedScenarios.get(2));

// if using test-bench which defines a simulation scenario
simpleDriver.runDiagnosis(ConsistencyType.INTERMITTENT);
```

Listing 3.13: Sample consistency driver

On the single simulation scenario user can employ {`STEP`, `PERSISTENT`, `INTERMITTENT`} diagnosis, where each type is defined as follows:

- **STEP** Diagnosis is performed at every time step
- **PERSISTENT** Diagnosis is performed after the simulation has been completed with assumptions that faults are persistent. Faulty components, if any, are returned.
- **INTERMITTENT** Diagnosis is performed after the simulation has been completed with assumptions that faults are intermittent. Faulty components, if any, are returned as well as time step at which faults manifested.

Construction concepts of both persistent fault and intermittent faults model are seen in Figure 3.5. Model of any system consists of health state variables

and logical sentences which describe their expected behavior as well as the behavior of the system.

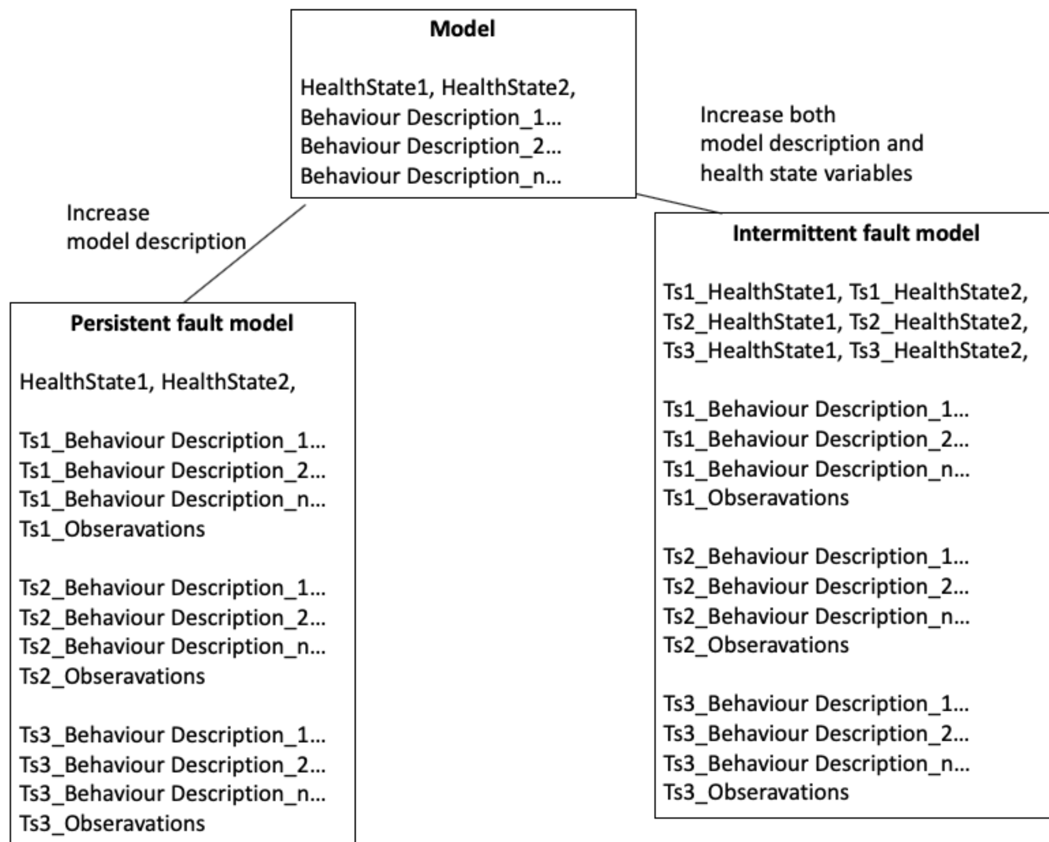

Figure 3.5.: Encoding of persistent and intermittent faults

In case of persistent faults, the model used for diagnosis consist of health state variables, model and observations, where model description and new observations are added at every time step to it while health state variables are constant (not incremented).
By the addition of system description with different variables describing its behavior, while health states are constant we will be able to determine diagnosis that explains all faults which happened thorough the run.
On the other hand, when assuming persistent faults health states are also

instantiated for every time step, to enable precise diagnosis. As new health states are added in every time step, the final number of health states to consider in diagnosis is multiple of the initial number of health states and time steps. Therefore, the intermittent diagnosis exponentially increased run time and memory requirements.

STEP	Persistent	Intermittent
Step 0 : [[]] Step 1 : [[AbRightWheel]] Step 2 : [[]] Step 3 : [[AbLeftWheel]] Step 4 : [[]]	[AbLeftWheel, AbRightWheel]	[AbLeftWheel_3, AbRightWheel_1]

Figure 3.6.: Output of different consistency based diagnosis procedures

As seen in Figure 3.6, `STEP` diagnosis produces a diagnosis for each step of the simulation, therefore for each step, RC-Tree is generated with the model and current observations. `PERSISTENT` and `INTERMITTENT`, however, generate RC-Tree just once, with model and observations designed as explained above.
Note that `INTERMITTENT` diagnosis procedure outputs a diagnosis where each health state variable is followed by a step in which it was diagnosed ($faultMode_{stepNumber}$).

The only difference between consistency and abductive driver is that for abductive diagnosis there is only one type of diagnosis present that returns diagnosis/explanations after every time step in the simulation, thus corresponding to `STEP` option of `ConsistencyDriver`. `AbductiveDriver::runDiagnosis` method only takes scenario as a parameter (or no parameter in case of $.fmi$ which defines a simulation scenario).

3.3. Back end

3.3.1. Submodules

PicoMUS

PicoMUS is a minimal unsatisfiable core extractor based on PicoSAT [1] SAT solver. In CatIO PicoMUS is used to check the consistency of the model itself, and in the case of inconsistencies, it is able to derive a minimal unsatisfiable subset.
Predicates of the model were previously mapped to integers to conform to the DIMACS standard. Minimal unsatisfiable subset consists of said integers. Integers that correspond to health state variables are extracted from the minimal unsatisfiable subset and used as labels in the RC-Tree algorithm.

Manual Scenario Generation

Simulation scenarios define the assignment of all input variables for the simulation. All input values have to be defined at step 0, and afterward, only those values which the user wants to change need to be defined. The rest of the input variables will stay the same. Scenarios are defined in `Scenario` class. In this class, the user defines the name of the scenario, initialization inputs (time step 0), and values for the user-defined time step. Those inputs are stored in the map with *key* being time step and value being list of `Component` objects which are going to be written to the simulation at *key* time step.

As seen in Table 3.1, for each scenario all input values for the initial time step (0) are provided. All values persist until the user changes them in subsequent time steps.

In the *intermittent* scenario found in the first 4 rows of Table 3.1, initialization is performed by providing values for all inputs at time step 0. Until time step 5 both wheels are spinning at the same speed and from time step 5 left wheel is spinning slower than the right as the left wheel voltage is decreased. The fault is injected in the time step 7 and the left wheel starts behaving

Table 3.1.: Defining several scenarios.

ScenarioID	Time Step	d	mode(Lw)	mode(Rw)	input(Lw)	input(Rw)
intermittent	0	2	ok	ok	4	4
	5				3	
	7			faster		
	15			ok		
noFaults	0	3	ok	ok	3	4
	3				4	3
	6				2	2
twoFaults	0	2	faster	ok	3	3
	10			slower		

abnormally. All other inputs are unchanged (left wheel input voltage is still 3 as defined in time step 5). Finally, at time step 15, the previously injected fault is corrected. At the lower table of Figure 3.7 several manually created in the graphical user interface scenarios can be seen.

Automated Scenario Generation

In CatIO, mixed level covering arrays defied in Section 2.7 are used for automatic generation of simulation scenarios which can be used either for individual testing or for automatic generation of abductive model.
Once `ModelData` has been constructed, it can be passed to `MCA` class in which user can experiment with several configuration options.

`MCA` class is based on ACTS [27] tool. ACTS is tool used for combinatorial test suite generation.

In `MCA` class following constraints over generation of the simulation suite can be defined:

- **Maximum number of faults injected** The upper limit of the number of fault modes for individual simulation scenarios that are faulty, that is not in the state *ok*
- **Fault Injection Step** Step of the simulation in which fault will be injected
- **Add relation to group** Relation in one of three groups (system inputs, parameters or mode assignment variables)
- **Add constraint** Constraint which hold for all rows in MCA

The maximum number of faults injected field is used to generate scenarios that would result in a "practical" diagnosis. Often designers are not interested in diagnosis which constrains more than 2 (sometimes 3) components. By setting the upper limit of fault components per simulation we reduce the final number of simulation scenarios created and if we do not add additional constraints we also assure that all combinations of faults up to a set limit are covered.

Constraints can be used to state physical impossibilities that would cause the simulation to misbehave/break. They can also be used to express dependencies/relations in the tree-structured systems [23]. If a fault is triggered in the component that is a parent of another component in which fault was also triggered, the second component's fault injection might be redundant if the behavior is completely dependant on the correct behavior of its parent.

A graphical user interface can be used to create a generated simulation suite (shown in Section 3.4). Generated simulation suite can be seen in the Listing 4.6.

Automatic Abductive Model Generator

Development of the diagnostic modeling is in practice often a tipping point that discourages engineers from using any kind of model-based reasoning or testing. In industry, however, simulation is often used as a way to ensure quality and reduce testing costs. Complex systems with its components are modeled in languages like Modelica. The principle idea of the automatic generation of the abductive diagnostic model is to use the already available simulation model and by the means of fault injection and

simulation generate the model which can be used for diagnostic reasoning. When injecting faults in components their effect on the system as a whole is observed.

The basic principle of the automatic generation of the abductive model is to compare fault free and fault injected simulation and detect discrepancies [16]. Such discrepancies are then encoded with respect to the model, in the form of *injectedFault(s)* $\rightarrow$ *detectedDiscrepancies*.
`Diff` interface is used to detect discrepancies between fault free and fault injected simulations. Furthermore, to decide which faults to inject, as well as which parameters and inputs are to be set in simulation, combination exploration of all possible model input combinations is used. Therefore before starting the automatic generation of the abductive model procedure, the user should create a combinatorial suite that defines simulation scenarios.

To automatically generate abductive model following methods are used

Listing 3.14: Automatic generation of abductive model setup

```
AbductiveModelGenerator abductiveModelGenerator =
new AbductiveModelGenerator(pathToRobotFmi, robotData, new
    RobotDiff());

\\ set encoder (not mandatory)
abductiveModelGenerator.setEnc(new StrongFaultAbEncoder());
\\ simulation runtime, step size and fault injection step
abductiveModelGenerator.generateModel(10, 1.0, 3);
\\ learn model
AbductiveModel learnedModel =
    abductiveModelGenerator.getAbductiveModel();

System.out.println(learnedModel.getRules());
learnedModel.modelToFile("test.txt");
```

Algorithm 2 presented in [17], is used for automatic generation of abductive model.

Algorithm 2 Algorithm for rule extraction with combinatorically explored input space and simulation

Require: A mixed level covering array MCA, simulation function $sim(inputs)$, implementation of $diff$

1: **procedure** GENERATE MODEL
2: $ruleSet \leftarrow \{\}$
3: **for** $simInputs \in MCA$ **do**
4: $faultFreeInput \leftarrow simInputs$
5: $faultInjectedInputs \leftarrow simInputs$
6: **for** $modeAssigmentVar \in faultFreeInput$ **do**
7: $modeAssigmentVar \leftarrow "Ok"$
8: **end for**
9: $Sim_{corr} \leftarrow sim(faultFreeInput)$
10: $Sim_{faulty} \leftarrow sim(faultInjectedInputs)$
11: $D \leftarrow diff(Sim_{corr}, Sim_{faulty})$
12: $injectedFaults \leftarrow \{\}$
13: **for** $modeAssigmentVar \in simInputs$ **do**
14: **if** $modeAssigmentVar \neq "Ok"$ **then**
15: $injectedFaults \leftarrow injectedFaults \cup modeAssigmentVar$
16: **end if**
17: **end for**
18: $newRule \leftarrow injectedFaults \rightarrow D$
19: $ruleSet \leftarrow ruleSet \cup newRule.$
20: **end for**
21: **return** $ruleSet$
22: **end procedure**

3.4. Graphical User Interface

Graphical user interface contains functionalities that ease tedious tasks of data extraction and modeling. It is separated in three sections: "Data and Simulation", which covers the extraction of data from .fmi files as well as generation of scenarios either manually or via mixed level covering array, "Consistency Modeling", which eases the task of modeling by allowing the user to experiment and shows all transformations which are done on the model and finally "Abductive Modeling' section where user can create and test abductive models. Let us discuss three sections of the graphical user interface in the following subsections.

3.4.1. Data extraction and scenario creators

In Figure 3.7 the user first selects a $.fmi$ file of the input-oriented model or test bench. Once the file has been selected, all fields of the model/test-bench will be displayed in the data extraction table.

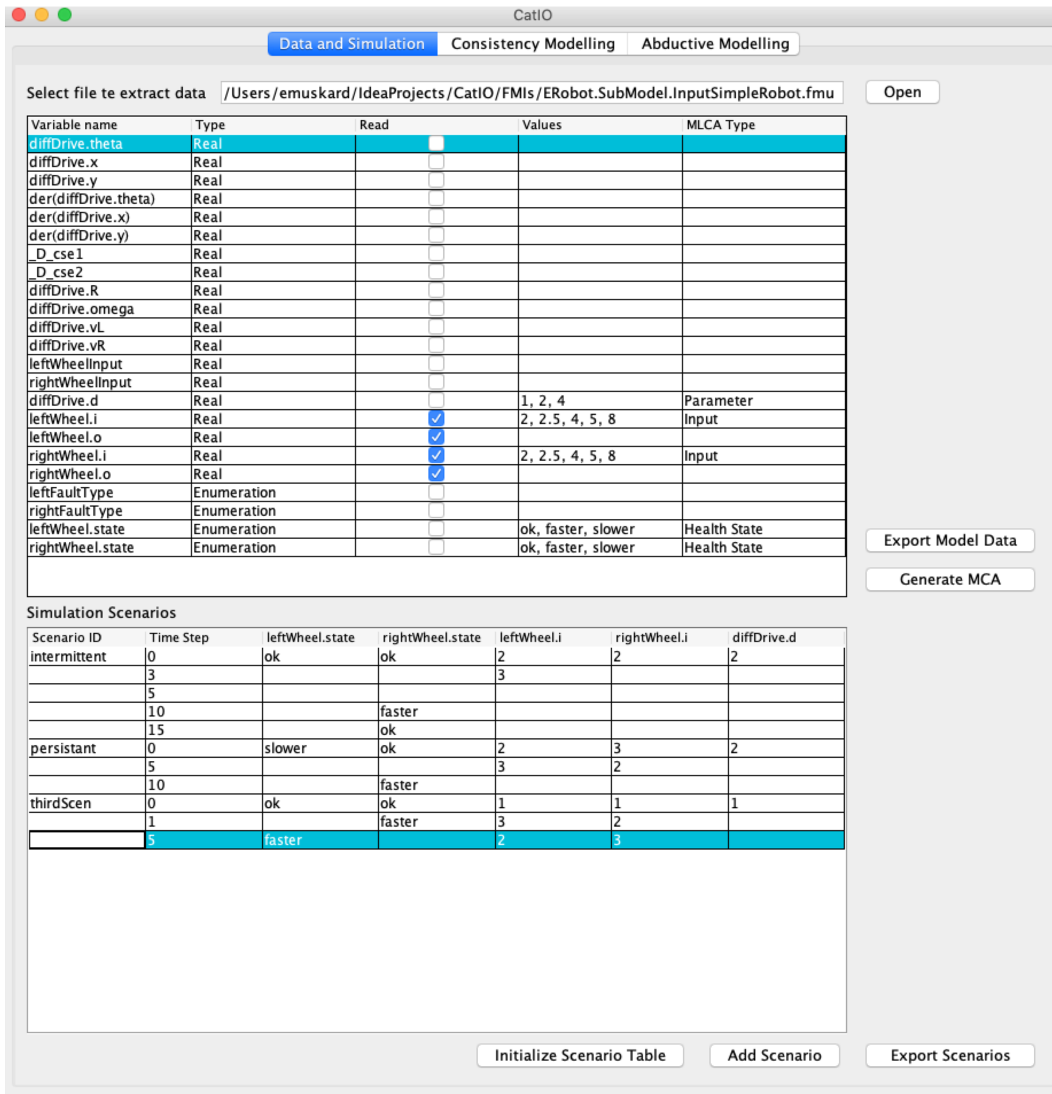

Figure 3.7.: Data extraction and simulation scenario creator window

Each field of the table is has a variable name (original name of the variable found in the Modelica model), type, and the following characteristic which the user can select:

- **Read** Variables whose values are going to be read and passed to the Encoder at every step of the simulation. Checked values will be added to the `ComponentsToRead` field of `ModelData`
- **Type** Type of variable, element of $\{HealthState, Input, Parameter\}$. Field, where a user selects a type, will appear in the table used for manual creation of scenarios. Used for automatic generation of simulation scenarios.
- **Values** Values are used in a mixed level covering array creation. Note that all value fields need to have an accompanying type.

On the press of the `Initialize scenario table` button scenario table where the user can create custom scenarios will be updated. As discussed in Section 3.3.1, all input values must have value at initialization. Afterward, at the chosen time step user defines inputs that are going to be changed.

If the user has provided values in the data extraction table, by pressing `Generate MCA` button a window shown in Figure 3.8 with several options will pop up. User can add relations, a step at which faults will be injected, constraints, and a number of correct components which will influence the generation of mixed level covering array as discussed in Section 3.3.1.

3.4.2. Modeling

Consistency-based and abductive diagnosis modeling environments can be used for the development of diagnostic models. Developed models can be manually verified by entering observations and computing the diagnosis.

In consistency modeling window, shown in Figure 3.9, two additional feature are present.
Firstly, user can see how CNF of the model, as well as mapping of, said CNF to DIMACS CNF format, which can be used for SAT solvers.
In the window under the CNF representation of the model, the user can see if the model itself (without observations) is satisfiable or unsatisfiable. If the model is satisfiable, an assignment of true/false to all propositions can be seen.
If the model is unsatisfiable, PicoMUS will compute minimal unsatisfiable

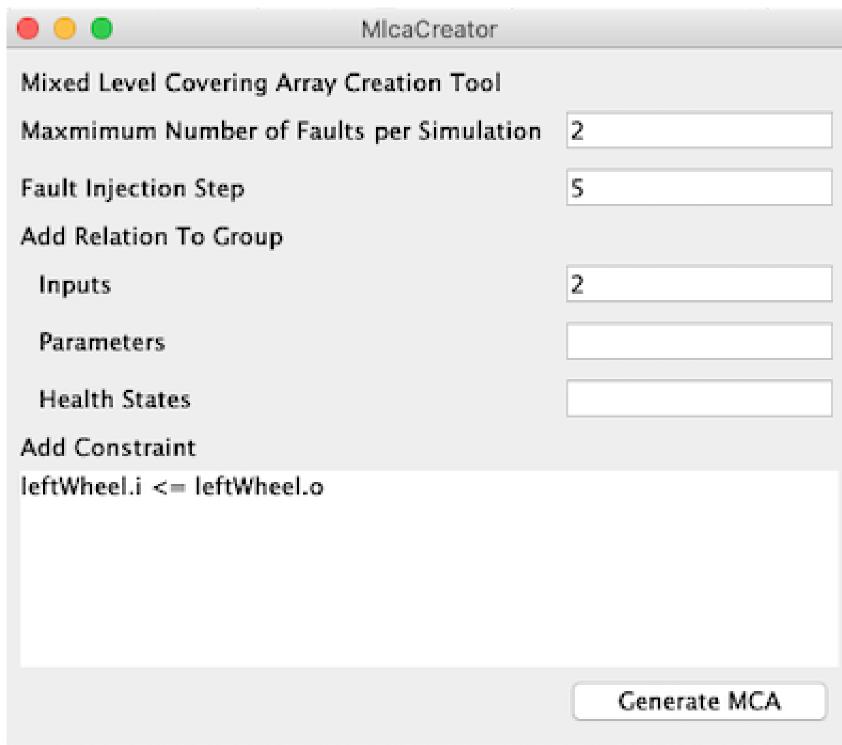

Figure 3.8.: Mixed level covering array generator window

core in a form of conjunction clauses that are still unsatisfiable. Those clauses will be printed and users may manually examine the cause of unsatisfiability.

Abductive modeling window can be seen if Figure 3.10.

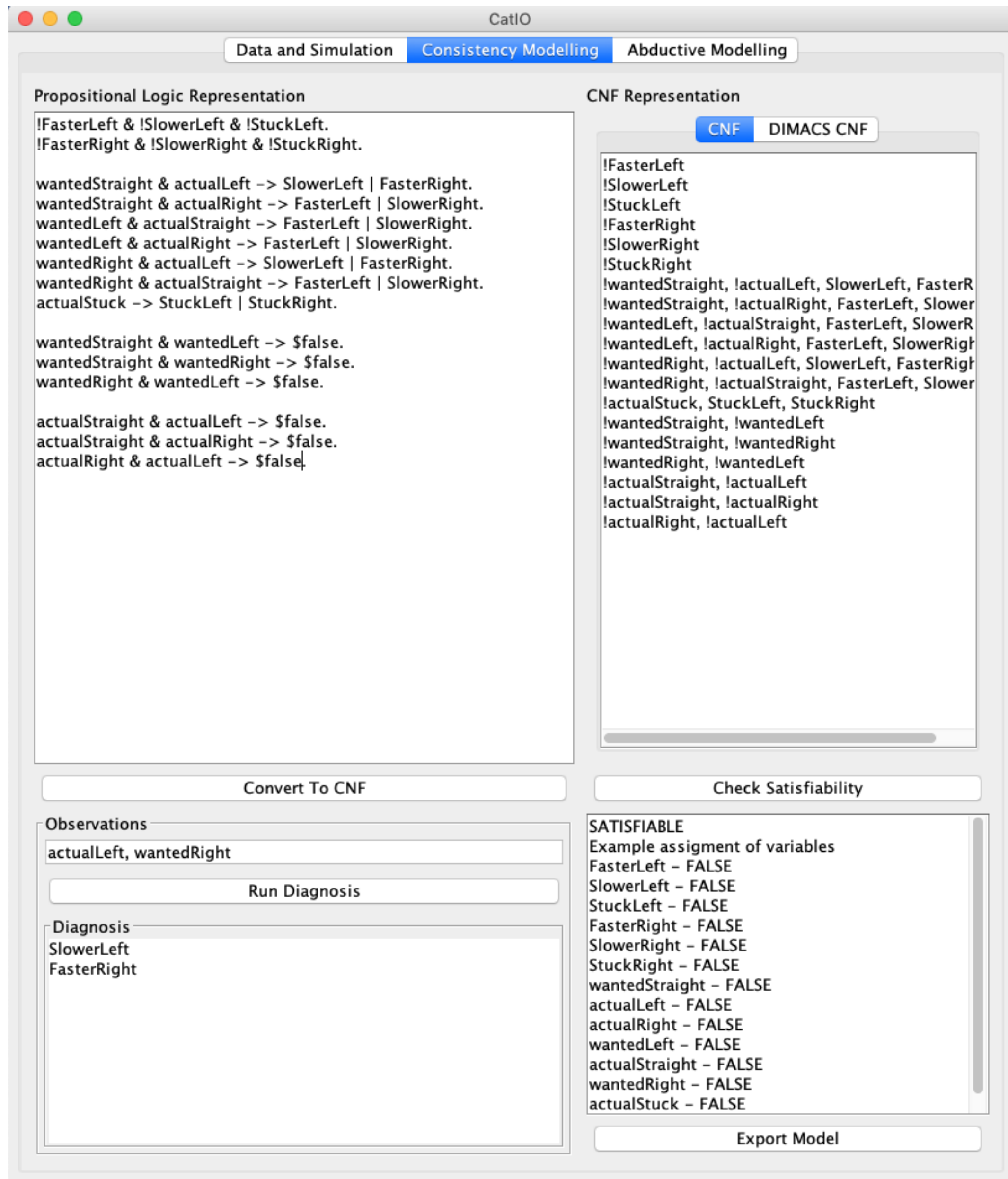

Figure 3.9.: Consistency based modeling window.

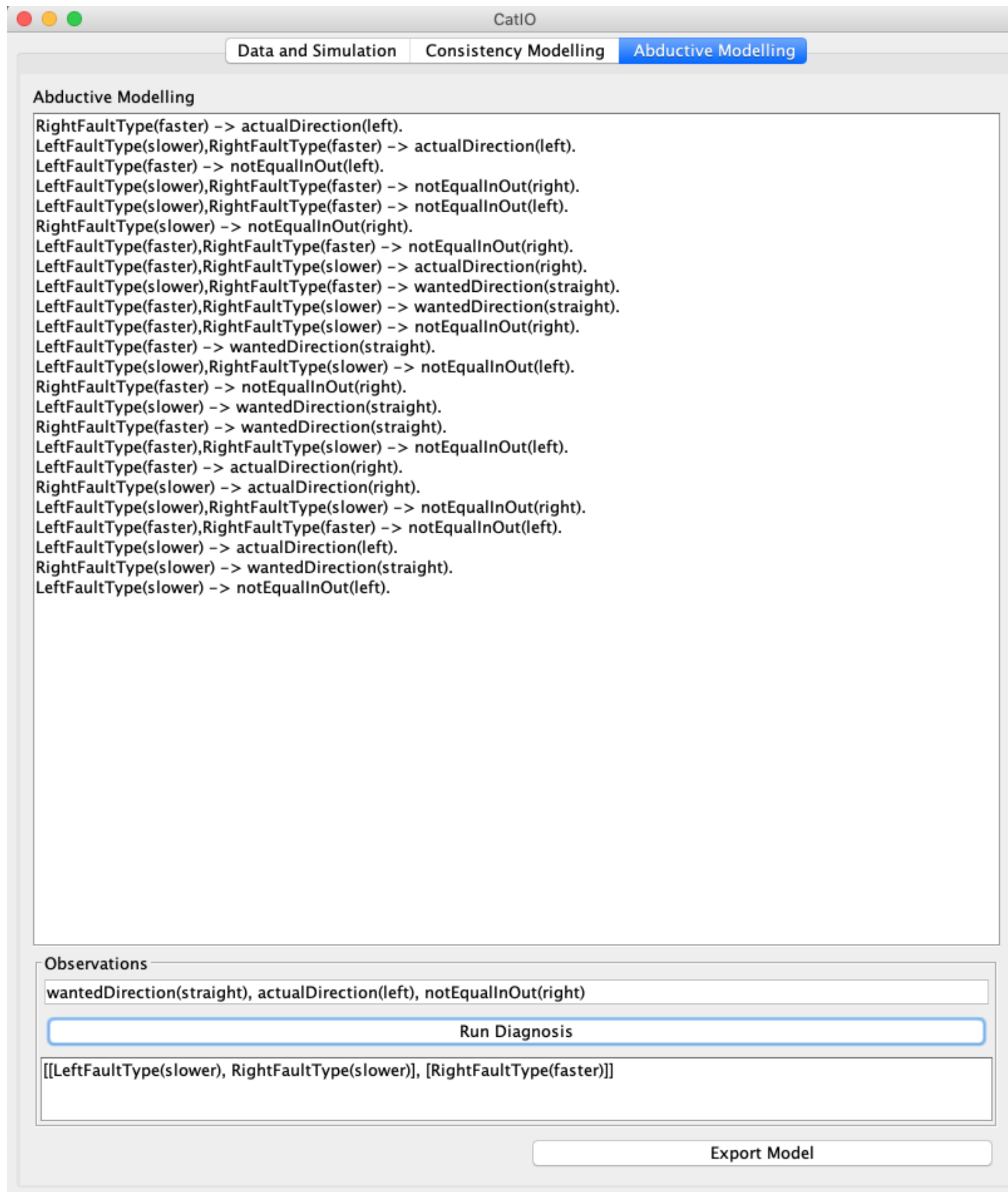

Figure 3.10.: Abductive modeling window

3.5. Code and Building/Running Instructions

Implementation of CatIO can be found at https://github.com/EdiMuskardin/CatIO-MBD-Framework.

CatIO is developed with Java 1.8 (Java 1.8.0_222). Gradle[1] build system is used to build and run the framework. In Figure 3.11 we can see the gradlew and gradlew.bat files that can be used to build and run the framework with only Java installed on the machine. The Gradle build system will automatically download all dependencies.

To build and run the framework one can use the following commands

- ./gradlew runGui to start the graphical user interface
- ./gradlew runDefaultMain to run ConsistenctMain example found in /src/examples folder.

By stating the path to the main class we may define additional run commands in the build.grade file.

Figure 3.11 depicts the structure of the CatIO found in the Github repository. README.md provides a quick summary of how to use the framework. In the *src/* folders and its subfolders implementation of all concepts explained in this chapter can be found.

To ensure that CatIO is working as expected, unit testing was performed on the individual component level. Furthermore, to validate the implementation of RC-Tree property-based testing (with Scalacheck) is used. Properties that have to hold for RC-Tree are that the output of the algorithm (diagnosis) is subset minimal and that each diagnosis is a hitting set for all inputs.

[1] https://gradle.org/

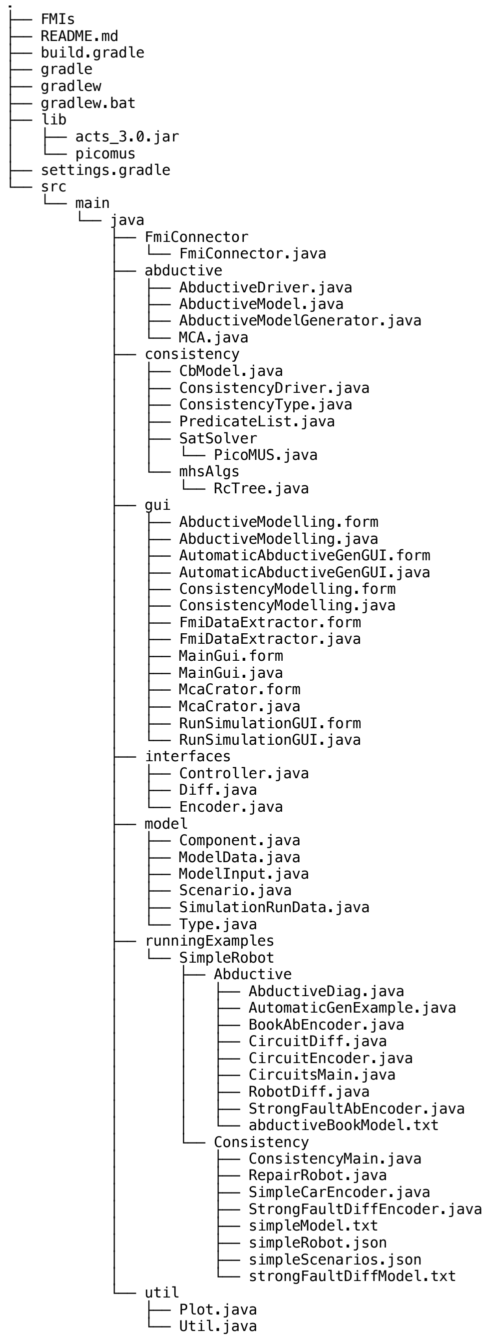

Figure 3.11.: CatIO's directory/file structure

4. Usage and Examples

In this chapter, we will demonstrate the capabilities of CatIO described in the previous chapter. We will focus mostly on the differential drive robot presented in Section 1.4. Several models and reasoning about their advantages and disadvantages are laid out.

Furthermore, we will present a simple model of an RC circuit with light bulbs of various sizes, to explore how well does to automatic generation of the abductive model scales.

4.1. Modeling the Differential Drive Robot

4.1.1. Modeling in Modelica

Figure 4.1 shows a differential drive robot with its connections between components as seen in the Modelica connection editor. u and $u1$ are standard Modelica notation for inputs, and they map to left wheel and right wheel input respectively. The output of both wheels is then passed to the $diffDrive$ component which computes the state of the robot.

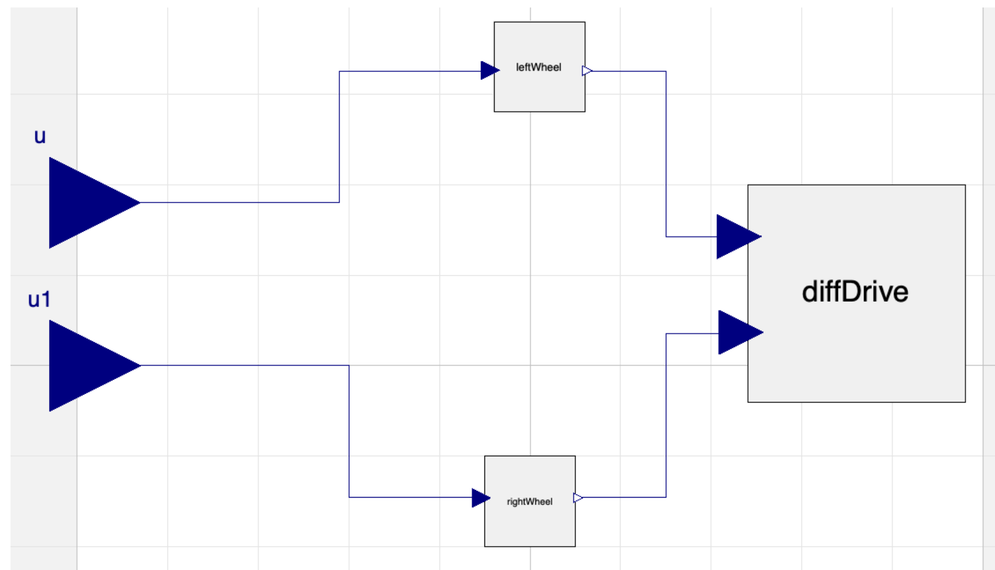

Figure 4.1.: Modelica model in connection editor

Wheels are modeled as a system with one input and one output. If the wheel is functioning as expected, input, which represents input voltage, should be the same as the output.
Fault injection is performed by changing the behavior of the wheel. Enumeration *FaultType* is created with 3 states: *ok*, *faster*, *slower*. If the wheel is behaving as expected it is in the *ok* state. When injecting a fault we change the value of the wheel's "state" variable from *ok* to *faster* or *slower*.

We modeled both faults by increasing or decreasing output voltage by 33% of the original one. We chose this approach instead of simply adding a constant to the output so that the fault scales with the size of the input.

Listing 4.1: Modelica model of the wheel with faults

```
model Wheel
  // initially wheel state is set to ok
  FaultType state(start = FaultType.ok);
  Modelica.Blocks.Interfaces.RealInput i;
  Modelica.Blocks.Interfaces.RealOutput o;
  equation
```

```
    if state == FaultType.ok then
      o = i;
    elseif state == FaultType.faster then
      o = i * 1.33;
    else
      o = i * 0.77;
    end if;
  end Wheel;
```

The behavior of the differential drive robot is explained in Section 1.4. Note that the position of the robot is calculated using differential equations, therefore $der(x)$ keyword is used. Five equations seen in *equations* block of Listing 4.2 describe the kinematics of the robot over time.

Listing 4.2: Modelica model of differential drive

```
model DiffDrive
  Modelica.Blocks.Interfaces.RealInput vL;
  Modelica.Blocks.Interfaces.RealInput vR;
  // distance between wheels
  parameter Modelica.SIunits.Distance d = 1;
  // x coordinate
  Modelica.SIunits.Position x(start = 0);
  // y coordinate
  Modelica.SIunits.Position y(start = 0);
  // state heading angle
  Modelica.SIunits.Angle theta(start = 0);
  // distance between the center of the robot and its
     inst. center of curvature
  Real R;
  // angular velocity
  Modelica.SIunits.AngularVelocity omega(start = 0,
     nominal = 1);
  equation
    R = d * (vL + vR) / 2 * (vL - vR);
    omega = (vR - vL) / 2;
    der(x) = (vR + vL) / 2 * cos(theta);
    der(y) = (vR + vL) / 2 * sin(theta);
    der(theta) = omega;
end DiffDrive;
```

Finally, we will use *input oriented model*, since we want to manually and automatically create scenarios and perform compensating/repair actions using CatIO.

Listing 4.3: Input oriented model of the robot

```
model InputSimpleRobot
  input WheelFaultType leftFaultType;
  input WheelFaultType rightFaultType;
  input Modelica.Blocks.Interfaces.RealInput
     leftWheelInput;
  input Modelica.Blocks.Interfaces.RealInput
     rightWheelInput;
  ERobot.SubModel.Wheel leftWheel;
  ERobot.SubModel.Wheel rightWheel;
  ERobot.SubModel.DiffDrive diffDrive;
equation
    leftWheel.state = leftFaultType;
    rightWheel.state = rightFaultType ;
    leftWheel.i = leftWheelInput;
    rightWheel.i = rightWheelInput;
    connect(leftWheel.o, diffDrive.vL);
    connect(rightWheel.o, diffDrive.vR);
end InputSimpleRobot;
```

4.1.2. Diagnosis and Repair

When constructing a diagnosis model of a system, we first have to determine which observations are available to us, as well as find the abstraction level of the model and observations.
We will create 3 consistency based models, with the following core principles:

- assumption that we can measure and compare inputs and outputs of each wheel
- assumption that we can observe both, the desired directions of the robot and actual one
- union of first two assumptions

Firstly, if we assume that we can conclude whether the input and output of the wheel are the same, but not strictly observe whether a wheel is spinning faster or slower we can create the following model.

Listing 4.4: Weak Fault Diagnostic Model

```
!AbLeftWheel -> leftEqInOut.
!AbRightWheel -> rightEqInOut.
```

From the model, it is obvious that if we get the observation $!leftlEqInOut$, which states that left wheel inputs and output are not the same, $AbLeftWheel$ will be the diagnosis.
With this model, we can only perform repair actions, as we are missing necessary information to perform any meaningful compensating action.

Suppose that we have the following scenario which we are going to simulate. The robot starts without any faults and begins to move in the straight line. At time step 5 of the simulation, the left wheel begins to spin faster, and this fault persists until time step 15. Such a scenario can be defined in the graphical user interface, by populating scenario creation table as shown In Table 4.1.
The goal of the robot is to move in the straight line, or at least to stay as close to the x axis as possible. x and y axis in the following images depict the movement of the robot thorough the time, If the robot is constantly staying on the x axis, he would be moving perfectly in the forward direction.

Table 4.1.: Single fault scenario.

ScenarioID	Time Step	d	mode(Lw)	mode(Rw)	input(Lw)	input(Rw)
intermittent	0	2	ok	ok	3	3
	5			faster		
	15			ok		

Simulation run time is defined at 20 time steps. Once a simulation has been terminated, we can plot the position of the robot in the coordinate plane. The number of plot points will be the same as the number of steps, in this case, 20. Note that the curve which describes the position of the robot is not smooth. If we were to take smaller steps size and more steps, the curve

would be smoother as there would be more plot points, but at the expanse of simulation speed.

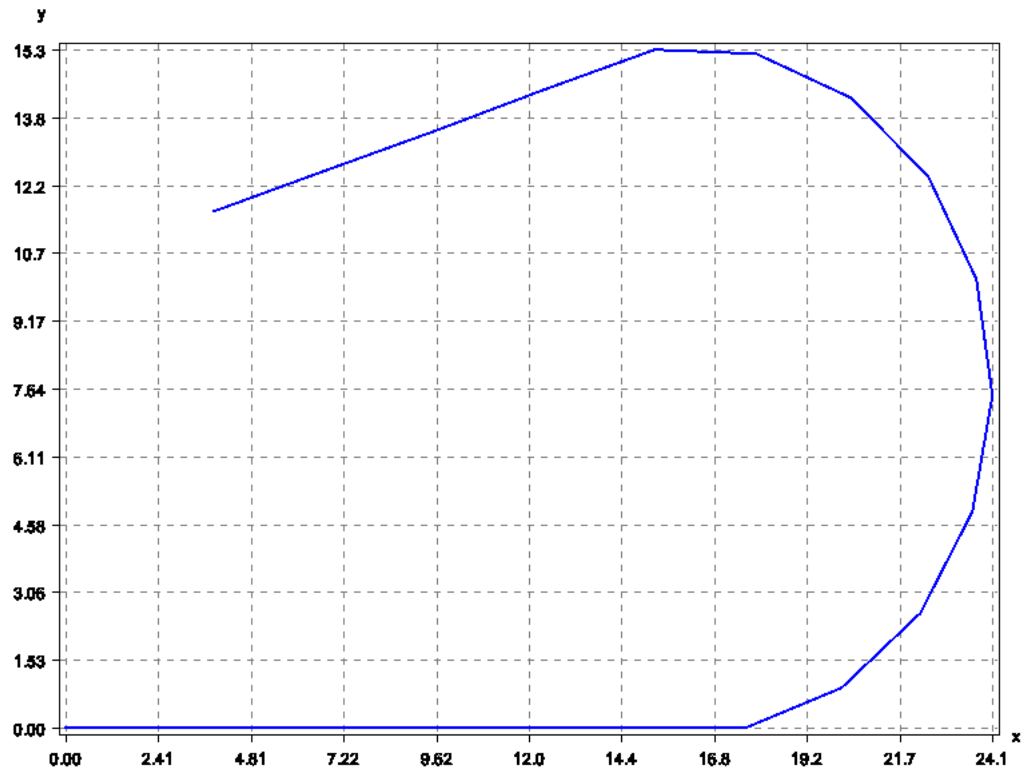

Figure 4.2.: Robot movement without any repair or compensating action

Suppose that by using a simple diagnostic model shown in 4.4, we can compute that the left wheel is behaving abnormally at time step 5. We do not know whether the left wheel is spinning faster or slower, but if we assume that we can put the wheel back to the correct state regardless of its current state, we can perform repair action.
As we can see in Figure 4.3, when performing the repair action the robot managed to stay closer to the x-axis and kept its direction straight, but the angle of its movement was still faulty. Between a detection of the fault and repair action, there has to be at least one step, as detection is only possible after a fault has occurred. Therefore heading angle change of that one step

is still a challenge.
However, in systems that are not time-dependent like the robot, repair actions would suffice to restore the behavior of the system to normal.

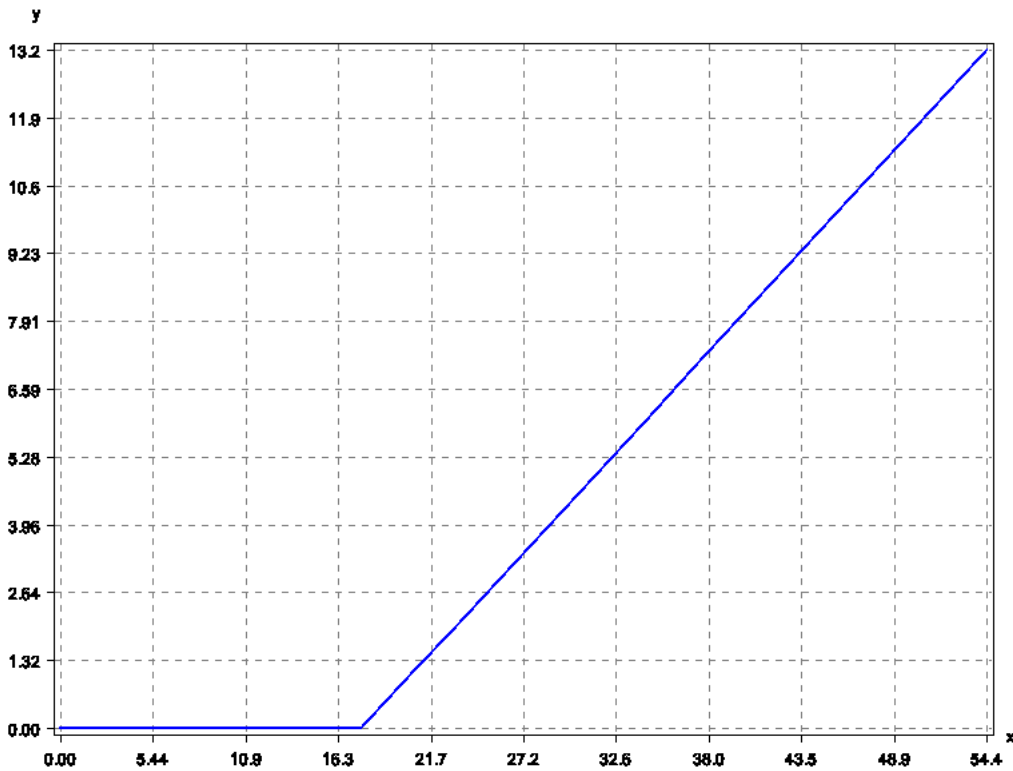

Figure 4.3.: Robot movement with repair action

Compensating actions can only be taken in case of diagnostic reasoning with the strong fault models. That is, without knowledge of a faulty component state, we cannot perform actions that would mitigate its consequences. In the Listing 4.5 we see the second model of differential drive robot where we assume that we only know desired and actual heading directions. The desired direction can be computed by comparing the inputs of both wheels, while the actual direction can be computed by comparing the outputs of both wheels. Note that the last 6 lines state physical impossibilities, like that it can not move in the 2 directions at the same time.

Listing 4.5: Strong Fault Diagnostic Model

```
wantedStraight & actualLeft -> SlowerLeft | FasterRight.
wantedStraight & actualRight -> FasterLeft | SlowerRight.
wantedLeft & actualStraight -> FasterLeft | SlowerRight.
wantedLeft & actualRight -> FasterLeft | SlowerRight.
wantedRight & actualLeft -> SlowerLeft | FasterRight.
wantedRight & actualStraight -> FasterLeft | SlowerRight.

// physical imposibilities
wantedStraight & wantedLeft -> $false.
wantedStraight & wantedRight -> $false.
wantedRight & wantedLeft -> $false.

actualStraight & actualLeft -> $false.
actualStraight & actualRight -> $false.
actualRight & actualLeft -> $false.
```

The obvious problem with this model is that a discrepancy in the movement direction can always be explained by either the left or right wheel. We can mitigate this problem by adding predicates which we used in the weak fault model (whether wheel input is the same as output) and all ambiguity would be gone.

For our example, if a minimal strong fault diagnosis has been computed it can be used to trigger one of the predefined compensating actions. In more complex examples depending on the result of the diagnosis and available data compensating actions first has to be derived. To compensate for the heading angle, we have to perform at least 2 compensating steps, 4 for optimal path correctness.
First, we set the speed of the opposite wheel for one time step by the same amount by which the faulty wheel has been increased, all while returning faulty wheel to correct state.
Then the heading angle is reversed back to the correct one, and we can set the input speed of both wheel to be the same. Once such compensating action has been performed, the plot of the robot movement is as seen in Figure 4.4.

Compensating action managed to correct the course of the robot completely. It was achieved in 4 steps, firstly by correcting the heading angle by increas-

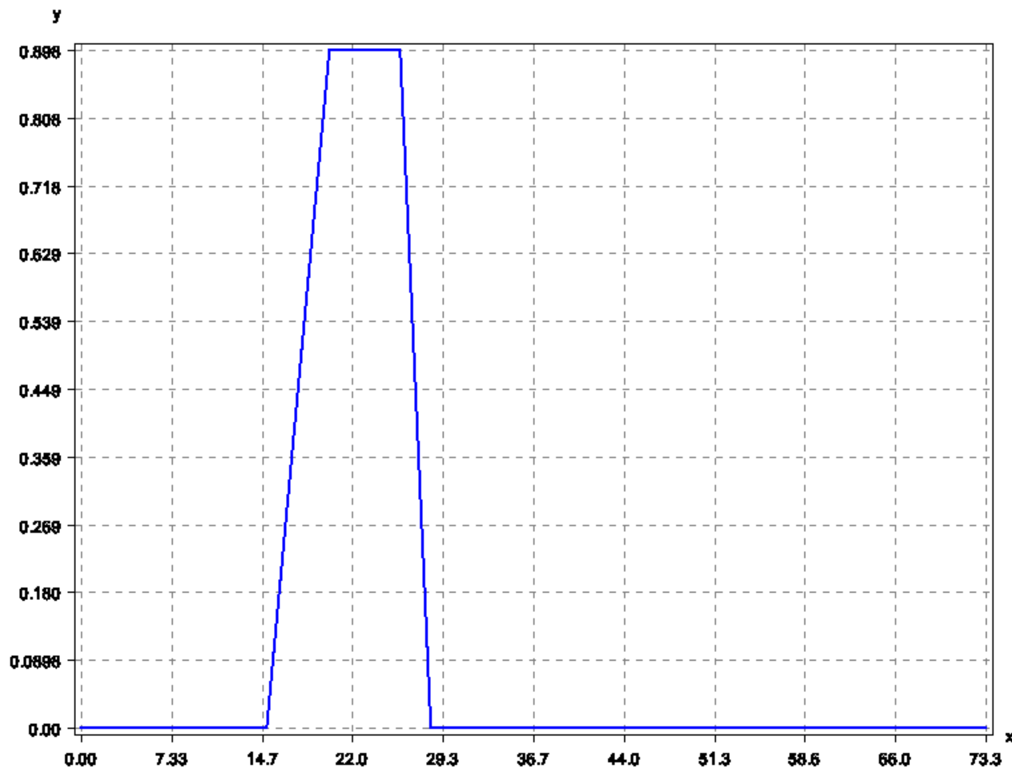

Figure 4.4.: Robot movement with compensating action

ing the input voltage of the right wheel while decreasing the left wheel input voltage. Once a heading direction is parallel with the intended one, inverse action can be taken to bring the robot back to the intended path.

To conclude, as seen in Figure 4.5 model-based diagnosis is a powerful tool that can be used to compute the appropriate decision needed to perform a repair or compensating action.

To be able to reproduce this example, one can use code found in the Appendix A.3.

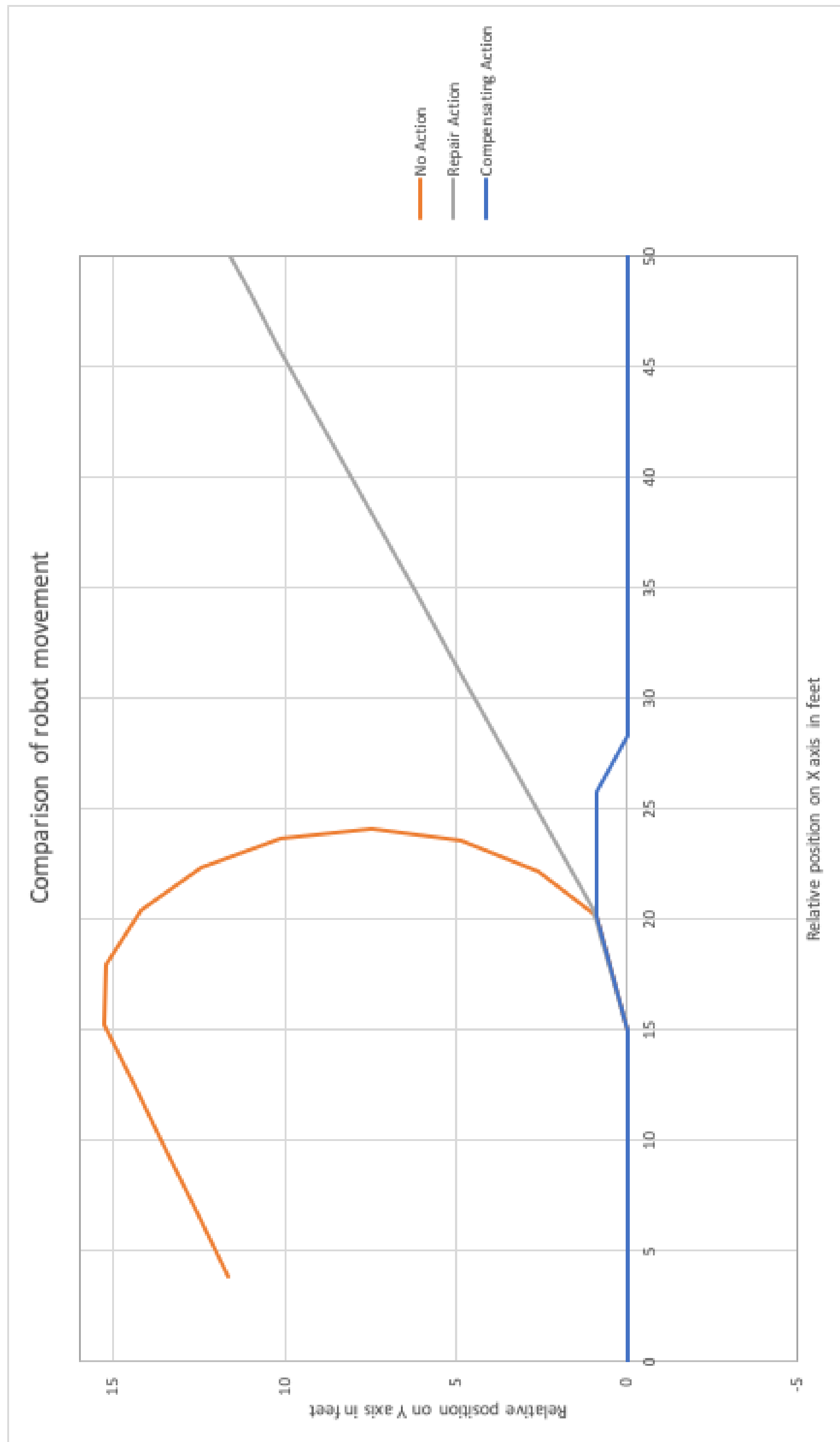

Figure 4.5.: Robot movement with repair and compensating actions

4.2. Automatic generation of abductive model

4.2.1. Differential Drive Robot

In the previous section, we developed a model and used the consistency-based diagnosis to get a meaningful diagnosis that we can use for repair or compensating actions. However, the task of modeling proves to be tedious in practice. Therefore in this section, we will demonstrate how to automatically generate the abductive model of the differential drive robot.

The level of abstraction will be the same as in the previous examples, and we will aim to learn a model which combines two approaches shown above, meaning we will use predicates to express whether the input and output of each wheel are the same and two predicates to express wanted and actual direction of the robot.

Firstly we extract `ModelData` values using a graphical user interface to be able to create a mixed-level coverings array.

# ACTS Test Suite Generation: Mon Jul 27 17:34:54 CEST 2020			
# '*' represents don't care value			
# Degree of interaction coverage: 2			
# Number of parameters: 4			
# Maximum number of values per parameter: 3			
# Number of configurations: 81			
leftFaultType	rightFaultType	leftWheelInput	rightWheelInput
ok	faster	2	2
ok	faster	2	4
ok	faster	2	8
ok	faster	8	8
ok	slower	2	2
ok	slower	2	4
ok	slower	2	8
ok	slower	4	2
ok	slower	4	4
ok	slower	4	8
ok	slower	8	2
ok	slower	8	4
ok	slower	8	8
faster	ok	2	2
faster	ok	2	4

Figure 4.6.: Automatically generated simulation scenarios.

Secondly, we first have to define encoder as seen in the Listing 4.6.

Listing 4.6: Encoder used for automatic generation of abductive model

```
public class StrongFaultAbEncoder implements Encoder {
@Override
public List<String> encodeObservation(Map<String, Object> obs) {
  List<String> encodedObservation = new ArrayList<>();
  Double rightWheelInput = (Double) obs.get("rightWheel.i");
  ...

  int wantedDir = Double.compare(rightWheelInput, leftWheelInput);
  int actualDir = Double.compare(rightWheelOutput, leftWheelOutput);

  if(wantedDir == 0)
    encodedObservation.add("wantedDirection(straight)");
  else if(wantedDir == 1)
        encodedObservation.add("wantedDirection(right)");
  else
        encodedObservation.add("wantedDirection(left)");

  if(actualDir == 0)
    encodedObservation.add("actualDirection(straight)");
  else if(actualDir == 1)
    encodedObservation.add("actualDirection(right)");
  else
    encodedObservation.add("actualDirection(left)");

  if (!rightWheelInput.equals(rightWheelOutput))
    encodedObservation.add("notEqualInOut(right)");
  else
    encodedObservation.add("EqualInOut(right)");

  if (!leftWheelInput.equals(leftWheelOutput))
    encodedObservation.add("notEqualInOut(left)");
  else
    encodedObservation.add("EqualInOut(left)");

  return encodedObservation;
  }}
```

Both fault free and fault injected simulation is executed and encoded observations are passed to the implementation of the `Diff` interface. A straightforward implementation of `Diff` interface can be seen in the Listing 4.7.

Listing 4.7: Implementation of Diff interface

```
public class RobotDiff implements Diff {

    @Override
    public Set<String> encodeDiff(SimulationRunData
        correct, SimulationRunData faulty) {
        Set<String> diff = new HashSet<>();
        for (int i = 0; i < correct.getNumberOfSteps();
            i++) {
            String corrWantedDir =
                correct.getPredicatesFromStep(i).get(0);
            // extraction of data...

            // check if actual directions of correct and
                faulty simulations are same
            if(!corrActualDir.equals(faultyActualDir)) {
                diff.add(corrWantedDir);
                diff.add(faultyActualDir);
            }
            // add a wheel which exerts different behavior
            if(!corrRightWheel.equals(faultyRightWheel))
                diff.add(faultyRightWheel);
            if(!corrLeftWheel.equals(faultyLeftWheel))
                diff.add(faultyLeftWheel);
        }
        return diff;
    }
}
```

`AbductiveModelGenerator` is used to connect all necessary inputs for the automatic generation of the model as shown in Listing 4.8.

Listing 4.8: Setup for learning of abductive model

```
ModelData robotData =
    Util.modelDataFromJson(pathToModelData);

AbductiveModelGenerator abductiveModelGenerator =
        new AbductiveModelGenerator(pathToRobotFmi,
            robotData);
abductiveModelGenerator.setEncoderAndDiff(new
    StrongFaultAbEncoder(), new RobotDiff());
// number of steps, step size, fault injection step
abductiveModelGenerator.generateModel(5, 1.0, 2);

AbductiveModel learnedModel =
    abductiveModelGenerator.getAbductiveModel();
learnedModel.modelToFile("learnedModel.txt");
learnedModel.addExplain(Arrays.asList("wantedDirection(straight)",
    "actualDirection(left)"));
System.out.println(learnedModel.getDiagnosis());
```

Once Algorithm 2 from Section 3.3.1 is executed, an automatically generated model can be saved to file and evaluated. This model is shown in the Listing 4.9.

Listing 4.9: Automatically generated abductive model

```
RightFaultType(faster) -> actualDirection(left).
LeftFaultType(slower),RightFaultType(faster) -> actualDirection(left).
LeftFaultType(faster) -> notEqualInOut(left).
LeftFaultType(slower),RightFaultType(faster) -> notEqualInOut(right).
LeftFaultType(slower),RightFaultType(faster) -> notEqualInOut(left).
RightFaultType(slower) -> notEqualInOut(right).
LeftFaultType(faster),RightFaultType(faster) -> notEqualInOut(right).
LeftFaultType(faster),RightFaultType(slower) -> actualDirection(right).
LeftFaultType(slower),RightFaultType(faster) -> wantedDirection(straight).
LeftFaultType(faster),RightFaultType(slower) -> wantedDirection(straight).
LeftFaultType(faster),RightFaultType(slower) -> notEqualInOut(right).
LeftFaultType(faster) -> wantedDirection(straight).
LeftFaultType(slower),RightFaultType(slower) -> notEqualInOut(left).
RightFaultType(faster) -> notEqualInOut(right).
LeftFaultType(slower) -> wantedDirection(straight).
RightFaultType(faster) -> wantedDirection(straight).
LeftFaultType(faster),RightFaultType(slower) -> notEqualInOut(left).
LeftFaultType(faster) -> actualDirection(right).
RightFaultType(slower) -> actualDirection(right).
LeftFaultType(slower),RightFaultType(slower) -> notEqualInOut(right).
LeftFaultType(faster),RightFaultType(faster) -> notEqualInOut(left).
LeftFaultType(slower) -> actualDirection(left).
RightFaultType(slower) -> wantedDirection(straight).
LeftFaultType(slower) -> notEqualInOut(left).
```

Depending on the observations generated model would compute the diagnosis. For example, observation

$$\{notEqualInOut(left)\}$$

would result in following diagnosis: {[[LeftFaultType(slower)], [LeftFaultType(faster)]]}, whereas observations

$$\{notEqualInOut(left), wantedDirection(straight), actualDirection(left)\}$$

would result in: {[[LeftFaultType(slower)], [LeftFaultType(faster), RightFaultType(faster)]]}.

Experiments presented in the previous section were repeated with automatically generated model and results were identical. Furthermore, this model can be extended by either adding rules about the behavior of the system or by stating physical impossibilities.

4.2.2. Automatic generation of RC-Circuits of various sizes

In this section, we will reason about the scalability of the automatic generation of the abductive model. The circuit is shown in Figure 4.8 shows a circuit which comprises of series of batteries, resistors, and light bulbs, all connected in parallel with the load. The load is represented as a series of the variable resistor and fixed value load resistor. Variable and load resistor ensures that the overall resistor value can not be zero, which would otherwise happen in the case that one of the resistors is shorted.

In the implementation, the values are set so that the bulbs are not always glowing. This is achieved by changing the load value of the r_{load} resistor. Its value follows the sinus curve over time. For a light bulb to be glowing we assume that the minimum current needs to pass through it.

The behavior of the simple electric circuits having only one battery and bulb is seen in Figure 4.7. *sut.r_load.percentage* gives the percentage of the resistor value of r_{load} over time. The other values are the current flowing through

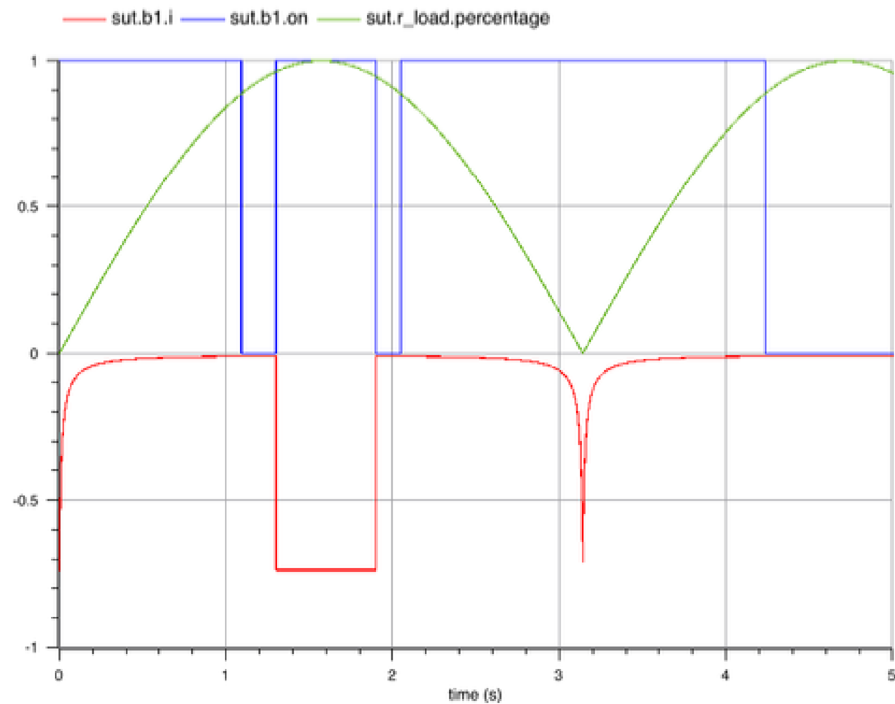

Figure 4.7.: Behavior of the simple electric circuits over time.

the bulb and information whether the bulb is lightning (1) or not lightning (0).

The size of the circuit can be arbitrary set, and we will explore how well does the automatic generation of diagnosis model fair on a simple circuit with just one series of battery, resistor, and light bulb, as well on the circuit with 10 parallel series.

Note that the only observation we can get from the running circuit (not in Modelica, but real-word scenario), is whether the bulb is shining or not, so this is the information with which we will work for our experiment.

`Encoder` and `Diff` interfaces can be reused for any circuit of size n, as only observations are n bulbs which are either on or off. In this example limitations of observations put a severe limitation on diagnosis quality, as

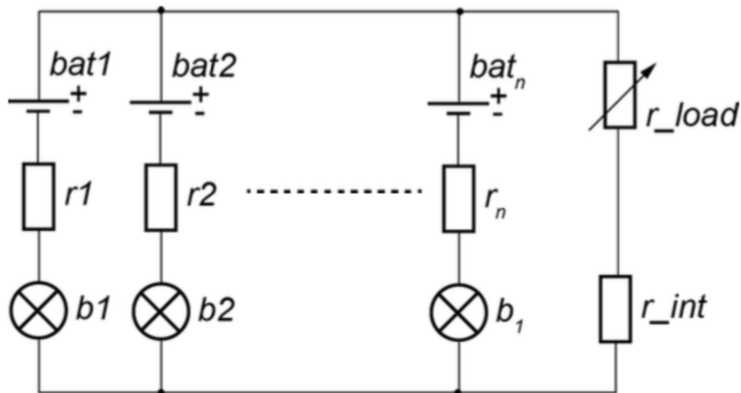

Figure 4.8.: Schematics of the simple electric circuits.

the only thing which can be compared in the `Diff` is whether light is on or off in both fault-free and fault injected simulation.

However, let us use this example to point out a few drawbacks when generating an automatic model for abductive diagnosis. As the size of the circuit will increase all possible combinations of components fault modes increase exponentially. We try to tackle that problem by using a combinatorial exploration of possible inputs or to be more precise in this example, of the fault modes. If a fault model of a component is never activated, it cannot be the result of the diagnosis as we do not have the information about its consequences and it will not appear in the knowledge base.

Following is a Modelica model of the circuit used, with each component having defined normal and faulty behavior. Listings 4.10 shows a circuit with one series of battery, resistor, and bulb, connected to the variable and load resistors. Presented circuits are analog with circuits that have more series of batteries, resistors, and bulbs.

Listing 4.10: Circuit with one battery resistor and bulb

```
model SC_Example1
  input PhysicalFaultModeling.FaultType bat1FaultState;
  input PhysicalFaultModeling.FaultType resistor1State;
  input PhysicalFaultModeling.FaultType bulb1State;
  input PhysicalFaultModeling.FaultType loadResistorState;
  input PhysicalFaultModeling.FaultType intResistorState;

  PhysicalFaultModeling.PFM_Battery bat1(vn = 4.5);
  PhysicalFaultModeling.PFM_Resistor r1(r = 0.1);
  PhysicalFaultModeling.PFM_Bulb b1(r=5);
  PhysicalFaultModeling.PFM_VariableResistor r_load(r =
     500);
  PhysicalFaultModeling.PFM_Resistor r_int(r=1);
  PhysicalFaultModeling.PFM_Ground gnd;

  equation
// connecting fault inputs to components
    r1.state = resistor1State;
    bat1.state = bat1FaultState;
    b1.state = bulb1State;
    r_load.state = loadResistorState;
    r_int.state = intResistorState;
    // load over time
    r_load.percentage = abs(sin(time));

    // connections of the circuit
    connect(bat1.m,r1.p);
    connect(r1.m,b1.p);
    connect(b1.m,gnd.p);
    connect(bat1.p,r_load.p);
    connect(r_load.m,r_int.p);
    connect(r_int.m,gnd.p);
end SC_Example1;
```

To keep the mixed level covering array smaller, we will assume that all the parameters of the components are constant. Extracting data is then done in GUI, as shown in the appendix A.3.

Implementation of `Encoder` and `Diff` interfaces is straightforward. `Encoder` interface is simply a mapping from Boolean domain to chosen predicates, and for bulb n in a circuit we encode either $bulb_n on$ or $bulb_n off$. `Diff` receives a list of lists which contain encoded information for every time step, and by comparing correct and faulty simulation we can conclude that bulb at step i is either on when it should be off and vice versa.
For this example, we can see that `Encoder` is redundant, as it is simply mapping Boolean values to predicates, and we can implement the `Diff` interface with values directly from the simulation.

Listing 4.11: Implementation of Diff

```
public class CircuitDiff implements Diff {
  @Override
  public Set<String> encodeDiff(List<List<String>> corr,
      List<List<String>> faulty) {
    Set<String> diff = new HashSet<>();

    for (int i = 0; i < corr.size(); i++) {
      for (int j = 0; j < corr.get(i).size(); j++) {
        String corrSim = corr.get(i).get(j);
        String faultySim = faulty.get(i).get(j);

        if(!corrSim.equals(faultySim)){
          if(corrSim.contains("_ON"))
            diff.add("bulb" + (j + 1) + "_OFF_INSTEAD_ON");
          else
            diff.add("bulb" + (j + 1) + "_ON_INSTEAD_OFF");
          }
        }
      }
  return diff;
  }
}
```

When learning a smaller circuit we can brute force all possible combinations of faults or up to n faults. To execute the automatic generation process code found in Appendix A.4 was executed. Part of a learned model for a circuit with one component set {bulb, resistor and battery} is shown in the Listing 4.12.

Listing 4.12: Learned circuit of size 1

```
Bat1.state(empty),R1.state(broken) -> bulb1_OFF_INSTEAD_ON.
B1.state(broken),R_load.state(broken) -> bulb1_ON_INSTEAD_OFF.
B1.state(short),R_int.state(broken) -> bulb1_OFF_INSTEAD_ON.
Bat1.state(empty),R_load.state(broken) -> bulb1_OFF_INSTEAD_ON.
R1.state(broken),R_load.state(broken) -> bulb1_ON_INSTEAD_OFF.
R_int.state(broken),R_load.state(broken) -> bulb1_ON_INSTEAD_OFF.
B1.state(short) -> bulb1_OFF_INSTEAD_ON.
...
```

The challenge of automatic generation of models is an exponential growth of possible combinations with respect to the states of the components and with the number of components. Even for small systems globally exhaustive approach is too costly. A globally exhaustive approach has to cover the product of the number of fault modes for all components. For a system comprising of 6 components, each with 3 fault modes one would have to cover $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 = 486$ combinations. If we would restrict ourselves to 2-way component interactions, the ACTS tool would compute 15 different scenarios for this model. 15 scenarios result in the simulation suite size reduction of 96.9%.

To demonstrate, let us consider the learned model of the circuit with 10 series of batteries, resistors, and bulbs. Suppose that for some component, eg. $battery_3$, we did not run a simulation where it states was set to empty. Therefore, $battery_3(empty)$ will never be a diagnosis, as it is not found in the generated model. We can mitigate this problem by stating that ACTS should generate a simulation suite with a maximum of 2 faults. We have 30 components, each with 3 fault modes, that would then result in $30 \times 3 = 90$ single fault and $\binom{90}{2}$ double fault simulations, resulting in 4095 simulation scenarios. If for each simulation we need half a second, we would be able to automatically generate the model in just above 30 minutes.

This example considers inputs and parameters constant. If we have to deal with a large number of parameters and input values we may experiment with different constraints and relations when creating the mixed level covering array which determines simulation parameters. Additionally, once a model has been generated we can manually extend it by adding rules which either further describe the model or state impossibilities.

5. Conclusion and Future Work

5.1. Conclusion

In this thesis, we presented a framework that supports designers in the implementation of model-based diagnosis for cyber-physical systems applications. Interaction with the Modelica simulation models allows the designer to cover a multitude of systems domains.

CatIO supports the designer in the process of developing models used for both consistency-based and abductive diagnosis. Consistency-based diagnosis models can be expressed by expressing the behavior of the system with propositional logic, whereas for the abductive models' rules are expressed as a set of Horn clauses by the PROLOG-like syntax. Developed models can be tested by providing observations manually or they can be tested by obtaining observations from the simulation scenarios. Automatic generation of simulation scenarios through the combinatorial exploration of all possible faults and inputs which are used in a concrete instance. By obtaining the observations from some Modelica simulation scenarios investigated with CatIO, a designer can quantitatively assess the diagnostic capabilities of a model early on during its development.

CatIO can interact with the simulation bidirectionally by reading and writing data of the simulation. From the simulations, observations used for diagnostic reasoning are obtained. Writing data to the simulation is used for the implementation of the external controller. This controller makes use of all available data (results of the diagnosis and current values of the simulation) and can react to them via dynamically defining the future inputs to the simulation. Concurrent development of both repair actions and diagnosis models can foster faster development of the diagnosis model which is able to compute diagnosis (with respect to the observations) which

contains enough information (about the faults) so that the controller can invoke the right actions.

Oftentimes the simulation models are available but the development of the diagnostic model from scratch may prevent the implementation of model-based diagnosis in a real-world project. Therefore CatIO can use the concept of the fault injection and simulation to automatically generate abductive models. Such automatically generated models can be investigated in Modelica simulations of the CPS to assess their diagnostic capabilities. Automatically generated models can be manually extended by stating physical impossibilities or additional behavior, which can lead to improved diagnostic capabilities.

The graphical user interface contains useful features that ease the development of diagnosis models as well as extraction of all needed data from the Modelica models. Furthermore, the manual and automatic creation of the simulation scenarios is supported by the graphical user interface.

Aside from CatIO, a framework that connects simulations and diagnosis is not known to the author and supervisors, and our architectural concept was positively accepted and it resulted in a publication "CatIO - A Framework for Model-Based Diagnosis of Cyber-Physical Systems" at 25th International Symposium on Methodologies for Intelligent Systems.

5.2. Future Work

The present version of CatIO is fully compatible with Modelica models, but since most modeling environments like MATLAB Simulink or Dymola provide an option to export the simulation/model to a functional mock-up interface, integration with other modeling languages should be straightforward. Special attention would have to be put to "input-oriented models", to check how to achieve the same functionality with other modeling languages.

In this thesis, a framework is presented in Java. However, an interesting approach would be to re-implement concepts presented in this thesis in Modelica itself (as an extension of OMEdit editor). OMEdit is a graphical

editor for Modelica. By adding the simple functionality of the encoder, which is easily achievable in Modelica, we could apply diagnostic reasoning to all simulations. Implementation of concepts found in this thesis can be done in the Modelica language itself. By doing so Modelica would have a self-contained environment used for the development of the simulations with model-based reasoning capabilities. The same holds for specific simulators used by the automotive or aerospace industry.

Extensions of modeling languages used for the creation of a diagnosis model may be considered. An example of the helpful extension of the modeling language would be the ability to define the behavior of each component with the function which defines inputs and outputs and their relation. This extension could shorten the development time of the diagnostic models and make them easier to understand.

CatIO's graphical user interface eases the tasks of data extraction, simulation generation, and modeling. An extension to the GUI in which a user can implement Encoder and Diff functions in a domain-specific language (to avoid the need for specific programming language) would make CatIO completely usable from GUI. Such an extension of the GUI would enable easier and faster validation of models, as well as ease the use of the framework for non-programmers.

Evaluation of the framework in a university course that deals with simulations and model-based diagnosis would provide valuable input to further improve the functionalities, documentation, and structure of the framework. Such evaluation was planned for the summer semester of 2020, but due to the extraordinary global situation, it was postponed.

Appendix

Appendix A.

Examples

A.1. ModelData from JSON file

```
{
  "componentsToRead":[
    {
      "name":"leftWheel.i",
      "type":"DOUBLE"
    },
    {
      "name":"leftWheel.o",
      "type":"DOUBLE"
    },
    {
      "name":"rightWheel.i",
      "type":"DOUBLE"
    },
    {
      "name":"rightWheel.o",
      "type":"DOUBLE"
    }
  ],
  "modeAssigmentVars":[
    {
      "name":"leftFaultType",
```

```
      "values":[
        "ok",
        "faster",
        "slower"
      ],
      "type":"ENUM",
      "originalName":"leftFaultType"
    },
    {
      "name":"rightFaultType",
      "values":[
        "ok",
        "faster",
        "slower"
      ],
      "type":"ENUM",
      "originalName":"rightFaultType"
    }
  ],
  "inputs":[
    {
      "name":"leftWheelInput",
      "values":[
        "2",
        "4",
        "8"
      ],
      "type":"DOUBLE",
      "originalName":"leftWheelInput"
    },
    {
      "name":"rightWheelInput",
      "values":[
        "2",
        "4",
        "8"
      ],
```

```
        "type":"DOUBLE",
        "originalName":"rightWheelInput"
      }
    ],
    "param":[

    ]
}
```


A.2. Simulation Scenario defined in JSON


```
{
    "scenarioId": "first",
    "timeCompMap": {
    "0": [
      {
        "name": "leftFaultType",
        "type": "ENUM",
        "value": "ok"
      },
      {
        "name": "rightFaultType",
        "type": "ENUM",
        "value": "ok"
},
{
        "name": "rightWheelInput",
        "type": "DOUBLE",
        "value": "3"
},
{
        "name": "leftWheelInput",
        "type": "DOUBLE",
        "value": "3"
```

```
}
],
  "5": [
  {
      "name": "leftFaultType",
      "type": "ENUM",
      "value": "faster"
  }
  ],
  "10": [
  {
      "name": "leftFaultType",
      "type": "ENUM",
      "value": "ok"
  }
  ]
  }
}
```

A.3. Diagnosis and Repair Example

Data and Simulation | Consistency Modelling | Abductive Modelling

Select file te extract data /Users/emuskard/IdeaProjects/CatIO/FMIs/SC_Example1.fmu Open

Variable name	Type	Read	Values	MLCA Type
bat1.p.i	Real			
r1.i	Real			
r1.m.i	Real			
r1.m.v	Real			
r1.p.i	Real			
r1.p.v	Real			
r_int.i	Real			
r_int.m.i	Real			
r_int.m.v	Real			
r_int.p.i	Real			
r_int.p.v	Real			
r_load.i	Real			
r_load.m.i	Real			
r_load.m.v	Real			
r_load.p.i	Real			
r_load.p.v	Real			
bat1FaultState	Enumeration		ok, empty	Health State
bulb1State	Enumeration		ok, short, broken	Health State
intResistorState	Enumeration		ok, short, broken	Health State
loadResistorState	Enumeration		ok, short, broken	Health State
resistor1State	Enumeration		ok, short, broken	Health State
b1.state	Enumeration			
bat1.state	Enumeration			
r1.state	Enumeration			
r_int.state	Enumeration			
r_load.state	Enumeration			
b1.on	Boolean	✓		

Export Model Data

Generate MCA

Figure A.1.: Extraction of data.

Listing A.1: Repair Example

```
public class RepairRobot implements Controller {
    private FmiConnector fmiConnector;
    private int leftFasterStep = 4;

    @Override
    public int performAction(FmiConnector fmiConnector,
       List<String> diagnosis) {
        this.fmiConnector = fmiConnector;
        for(String diag : diagnosis) {
            if (diag.equals("AbLeftWheel"))
                return repairLeftWheel();
            if (diag.equals("AbRightWheel"))
```

```
                return repairRightWheel();
            if (diag.equals("faster(leftWheel)")){
                return compensateLeftFaster();
            }
            /// ...
        }
        return compensateLeftFaster();
    }

    private int repairLeftWheel(){
        // to repair a component we simply put it state to
           "ok", which always corresponds to 1
        fmiConnector.writeVar("leftFaultType", 1);
        return 0;
    }

    private int repairRightWheel(){
        fmiConnector.writeVar("rightFaultType", 1);
        return 0;
    }

    private int compensateLeftFaster(){
        // notice that in this example instead of writing
           values which correspond to wheel speed
        // we can use wheel fault types to achive same effect,
           of one wheel spinning faster, other slower etc.
        // note that 1 always corresponds to ok state, 2 is
           faster, and 3 is slower
        if(leftFasterStep == 4) {
            fmiConnector.writeVar("leftFaultType", 1);
            fmiConnector.writeVar("rightFaultType", 2);
            return --leftFasterStep;
        }
        else if(leftFasterStep == 3){
            fmiConnector.writeVar("leftFaultType", 1);
            fmiConnector.writeVar("rightFaultType", 1);
            return --leftFasterStep;
        }
        else if(leftFasterStep == 2){
```

```
            fmiConnector.writeVar("leftFaultType", 1);
            fmiConnector.writeVar("rightFaultType", 2);
            return --leftFasterStep;
        }else if(leftFasterStep == 1){
            fmiConnector.writeVar("leftFaultType", 2);
            fmiConnector.writeVar("rightFaultType", 1);
            return --leftFasterStep;
        }
        else{
            // finnaly set wheels to correct state
            repairRightWheel();
            repairLeftWheel();
            return --leftFasterStep;
        }
    }
}
```

Listing A.2: Main class for repair example

```
public class ConsistencyMain {
    public static void main(String[] args) {
        String inputModelRobot =
           "FMIs/ERobot.SubModel.InputSimpleRobot.fmu";
        String pathToSimpleModel = "simpleModel.txt";
        String pathToScenarios = "simpleScenarios.json";
        String pathToModelData = "simpleRobot.json";

        // Robot data extracted from JSON
        ModelData modelData = Util.modelDataFromJson(pathToModelData);
        // Set values to be plotted
        modelData.setPlotVariables("diffDrive.x", "diffDrive.y");
        // Set controller which will perform repair actions
        modelData.setController(new RepairRobot());
        // Connect everything together
        ConsistencyDriver consistencyDriver =
           ConsistencyDriver.builder()
                 .pathToFmi(inputModelRobot)
                 .model(new CbModel(pathToSimpleModel))
                 .encoder(new SimpleCarEncoder())
                 .modelData(modelData)
```

```
                    .numberOfSteps(20)
                    .simulationStepSize(1)
                    .build();

            // Load simulations from file
            List<Scenario> scenarios =
                Util.scenariosFromJson(pathToScenarios, modelData);
            // Run diagnosis and if possible repair from scenario 0
            consistencyDriver.runDiagnosis(ConsistencyType.STEP,
                scenarios.get(0));
        }
}
```

Listing A.3: Encoder and Diff for automatic generation of abductive model.

```
public class StrongFaultAbEncoder implements Encoder {
    @Override
    public List<String> encodeObservation(Map<String, Object> obs)
        {
            List<String> encodedObservation = new ArrayList<>();
            Double rightWheelInput = (Double) obs.get("rightWheel.i");
            Double rightWheelOutput = (Double) obs.get("rightWheel.o");
            Double leftWheelInput = (Double) obs.get("leftWheel.i");
            Double leftWheelOutput = (Double) obs.get("leftWheel.o");

            int wantedDir = Double.compare(rightWheelInput,
                leftWheelInput);
            int actualDir = Double.compare(rightWheelOutput,
                leftWheelOutput);

            if(wantedDir == 0)
                encodedObservation.add("wantedDirection(straight)");
            else if(wantedDir == 1)
                encodedObservation.add("wantedDirection(right)");
            else
                encodedObservation.add("wantedDirection(left)");

            if(actualDir == 0)
                encodedObservation.add("actualDirection(straight)");
            else if(actualDir == 1)
```

```
            encodedObservation.add("actualDirection(right)");
        else
            encodedObservation.add("actualDirection(left)");

        if(!rightWheelInput.equals(rightWheelOutput))
            encodedObservation.add("notEqualInOut(right)");
        else
            encodedObservation.add("EqualInOut(right)");

        if(!leftWheelInput.equals(leftWheelOutput))
            encodedObservation.add("notEqualInOut(left)");
        else
            encodedObservation.add("EqualInOut(left)");

        return encodedObservation;
    }
}

public class RobotDiff implements Diff {
    @Override
    public Set<String> encodeDiff(List<List<String>> corr,
       List<List<String>> faulty) {
        Set<String> diff = new HashSet<>();
        for (int i = 0; i < corr.size(); i++) {
            // as seen in encoder, predicates are
            // wanted direction
            // actual direction
            // is left wheel input equal to output
            // is right wheel input equal to output
            String corrWantedDir = corr.get(i).get(0);
            String corrActualDir = corr.get(i).get(1);
            String corrRightWheel = corr.get(i).get(2);
            String corrLeftWheel = corr.get(i).get(3);

            String faultyWantedDir = faulty.get(i).get(0);
            String faultyActualDir = faulty.get(i).get(1);
            String faultyRightWheel = faulty.get(i).get(2);
            String faultyLeftWheel = faulty.get(i).get(3);
```

```
                // check if actual directions of correct and faulty
                   simulations are same
                if(!corrActualDir.equals(faultyActualDir)) {
                    diff.add(corrWantedDir);
                    diff.add(faultyActualDir);
                }
                // add a wheel which exerts different behaviour
                if(!corrRightWheel.equals(faultyRightWheel))
                    diff.add(faultyRightWheel);
                if(!corrLeftWheel.equals(faultyLeftWheel))
                    diff.add(faultyLeftWheel);
            }
            return diff;
        }
    }
```

A.4. Circuit Encoder and Diff Implementation

Listing A.4: Implementation of Encoder

```
public class CircuitEncoder implements Encoder {
  @Override
  public List<String> encodeObservation(Map<String, Object> obs) {
    List<String> observations = new ArrayList<>();
    for (int i = 0; i < obs.values().size(); i++) {
      String bulbId = "b" + (i + 1) + ".on";
        if ((Boolean) obs.get(bulbId)) {
          observations.add("bulb" + (i + 1) + "_ON");
      } else {
          observations.add("bulb" + (i + 1) + "_OFF");
        }
      }
    return observations;
  }
}
```

Listing A.5: Example of automatic generation of diagnostic model

```
public class CircuitsMain {
  public static void main(String[] args) throws IOException {
    String pathToCircuitFmi = "FMIs/SC_Example1.fmu";
    String pathToModelData = "singleCircuit.json";
    ModelData circuitData = Util.modelDataFromJson(pathToModelData);

    AbductiveModelGenerator abductiveModelGenerator = new
        AbductiveModelGenerator(pathToCircuitFmi, circuitData, new
        CircuitDiff());
    abductiveModelGenerator.setEnc(new CircuitEncoder());

    // mixed level covering array will be automatically generated
        without any constraints
    // params for generate model are number of steps, step size and
        fault injection step
    abductiveModelGenerator.generateModel(50, 0.5, 20);
    AbductiveModel learnedModel =
        abductiveModelGenerator.getAbductiveModel();
```

```
        System.out.println(learnedModel.getRules());
        learnedModel.modelToFile("singleCircuit.txt");
    }
}
```

Bibliography

[1] Armin Biere. "PicoSAT Essentials." In: *JSAT* 4 (May 2008), pp. 75–97. DOI: 10.3233/SAT190039 (cit. on p. 45).

[2] T. Blochwitz et al. "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models." In: Sept. 2012. DOI: 10.3384/ecp12076173 (cit. on p. 27).

[3] Luca Console and Pietro Torasso. "Integrating Models of Correct Behavior into Abductive Diagnosis." In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Stockholm: Pitman Publishing, Aug. 1990, pp. 160–166 (cit. on p. 19).

[4] Franz Wotawa Edi Muškardin Ingo Pill. "CatIO - A Framework for Model-Based Diagnosis of Cyber-Physical Systems." In: 2020 (cit. on p. 2).

[5] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. "Hypothesis Classification, Abductive Diagnosis and Therapy." In: *Proceedings of the International Workshop on Expert Systems in Engineering*. Vienna: Springer Verlag, Lecture Notes in Artificial Intelligence, Vo. 462, Sept. 1990 (cit. on p. 19).

[6] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons, Dec. 2014 (cit. on pp. 3, 29).

[7] Christopher S. Gray et al. "An Abductive Diagnosis and Modeling Concept for Wind Power Plants." English. In: *International Workshop on Principles of Diagnosis*. ., 2014, pp. 404–409 (cit. on p. 2).

[8] R. Greiner, B. A. Smith, and R. W. Wilkerson. "A Correction to the Algorithm in Reiter's Theory of Diagnosis." In: *Artificial Intelligence* 41.1 (1989), pp. 79–88 (cit. on p. 17).

[9] J. de Kleer and B. C. Williams. "Diagnosing Multiple Faults." In: *Artificial Intelligence* 32.1 (1987), pp. 97–130 (cit. on pp. 11, 14).

[10] Benjamin Kuipers and Jerome P. Kassirer. "Causal Reasoning in Medicine: Analysis of a Protocol." In: *Cognitive Science* 8.4 (1984), pp. 363–385. DOI: 10.1207/s15516709cog0804_3. URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog0804_3 (cit. on p. 13).

[11] A. Metodi et al. "Compiling Model-Based Diagnosis to Boolean Satisfaction." In: *AAAI Conference on Artificial Intelligence*. 2012, pp. 793–799 (cit. on p. 16).

[12] I. Nica et al. "The Route to Success - A Performance Comparison of Diagnosis Algorithms." In: *International Joint Conference on Artificial Intelligence (IJCAI)*. 2013, pp. 1039–1045 (cit. on p. 16).

[13] I. Pill and T. Quaritsch. "An LTL SAT Encoding for Behavioral Diagnosis." In: *Int. Workshop on the Principles of Diagnosis*. 2012 (cit. on p. 15).

[14] I. Pill and T. Quaritsch. "RC-Tree: A variant avoiding all the redundancy in Reiter's minimal hitting set algorithm." In: *IEEE Int. Symp. Software Reliability Engineering Workshops (ISSREW)*. 2015, pp. 78–84. DOI: 10.1109/ISSREW.2015.7392050 (cit. on pp. 17, 18).

[15] I. Pill, T. Quaritsch, and F. Wotawa. "From Conflicts to Diagnoses: An Empirical Evaluation of Minimal Hitting Set Algorithms." In: *22nd Int. Workshop on the Principles of Diagnosis*. 2011, pp. 203–210 (cit. on p. 18).

[16] Ingo Pill and Franz Wotawa. "Fault detection and localization using Modelica and abductive reasoning." In: *Diagnosability, Security and Safety of Hybrid Dynamic and Cyber-Physical Systems*. 2018, pp. 45–72. ISBN: 9783319749617 (cit. on pp. 22, 48).

[17] Ingo Pill and Franz Wotawa. "Model-Based Diagnosis Meets Combinatorial Testing For Generating an Abductive Diagnosis Model." English. In: *28th International Workshop on Principles of Diagnosis (DX'17)*. Kalpa Publications in Computing. United Kingdom: EasyChair Ltd, Jan. 2018, pp. 248–263. DOI: 10.29007/svc7 (cit. on p. 49).

[18] "Predicate Logic." In: *Logic for Computer Scientists*. 2008, pp. 41–107 (cit. on p. 8).

[19] R. Reiter. "A Theory of Diagnosis from First Principles." In: *Artificial Intelligence* 32.1 (1987), pp. 57–95 (cit. on pp. 11, 15).

[20] Raymond Reiter and Johan Kleer. "Foundations of Assumption-based Truth Maintenance Systems: Preliminary Report." In: Jan. 1987, pp. 183–189 (cit. on p. 20).

[21] Stuart J. Russell and Peter Norvig. "Artificial Intelligence: A Modern Approach." In: 1995 (cit. on p. 33).

[22] Moamar Sayed-Mouchaweh. *Diagnosability, Security and Safety of Hybrid Dynamic and Cyber-Physical Systems*. 1st. Springer Publishing Company, Incorporated, 2018. ISBN: 3319749617 (cit. on p. 1).

[23] Markus Stumptner and Franz Wotawa. "Diagnosing tree-structured systems." In: *Artificial Intelligence* 127.1 (2001), pp. 1–29. ISSN: 0004-3702 (cit. on p. 47).

[24] Jeffrey M. Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs against Errors*. John Wiley & Sons, Inc., 1997 (cit. on p. 22).

[25] Andrew Whitmore, Anurag Agarwal, and Lida Xu. "The Internet of Things—A survey of topics and trends." In: *Information Systems Frontiers* 17 (2015), pp. 261–274 (cit. on p. 1).

[26] Franz Wotawa. "Reasoning from first principles for self-adaptive and autonomous systems." In: *Predictive Maintenance in Dynamic Systems – Advanced Methods, Decision Support Tools and Real-World Applications*. Ed. by E. Lughofer and M. Sayed-Mouchaweh. Springer, 2019. DOI: 10.1007/978-3-030-05645-2 (cit. on p. 2).

[27] L. Yu et al. "ACTS: A Combinatorial Test Generation Tool." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pp. 370–375. DOI: 10.1109/ICST.2013.52 (cit. on p. 46).