9

A Framework for Integrating Automated Diagnosis into Simulation

David Kaufmann and Franz Wotawa

Graz University of Technology, Austria

Abstract

Automatically detecting and locating faults in systems is of particular interest for mitigating undesired effects during operation. Many diagnosis approaches have been proposed including model-based diagnosis, which allows to derive diagnoses from system models directly. In this paper, we present a framework bringing together simulation models with diagnosis allowing for evaluating and testing diagnosis models close to its real world application. The framework makes use of functional mock-up units for bringing together simulation models and enables their integration with ordinary programs written in either Python or Java. We present the integration of simulation and diagnosis using a two-lamp example model.

Keywords: model-based diagnosis, fault detection, fault localization, physical simulation.

9.1 Introduction

To keep systems operational, we need to carry out diagnoses regularly. Diagnosis includes the detection of failures, the localization of corresponding root causes, and repair. We carry out regular maintenance activities that include diagnosis and predictions regarding the remaining lifetime of components to prevent systems from breaking during use. However, there is no guarantee

that system components are not breaking during operation, even when carrying out maintenance as requested. In some cases, it is sufficient to indicate such a failure, i.e., via presenting a warning or error message and passing mitigation measures to someone else. Unfortunately, there are systems like autonomous systems where we can hardly achieve such a mitigation process. For example, in fully autonomous driving, there is no driver anymore for passing control. Therefore, there is a need for coming up with advanced diagnosis solutions that cover detection, localization, and repair. A practical real world problem demonstration of an on-board control agent was validated in the year 1999, within the scope of Deep Space One, a space exploration mission, carried out by NASA. Regarding this, the authors of the paper [4] describe developed methods related to model-based programming principles, including the area of model-based diagnosis. The methods were applied on autonomous systems, designed for high reliability, operating as subject of a spacecraft system.

When we want to integrate advanced diagnosis into systems, we need to come up with means for allowing us to easily couple monitoring with diagnosis. As stated by the authors in [3], the coupling enables the diagnosis method to detect and localize faults based on observations, obtained by monitoring a cyber-physical system (CPS). Furthermore, we require close integration of today's development processes, which rely on system simulation. The latter aspect is of uttermost importance for showing early that diagnosis based on monitoring can improve the overall behaviour of a system even when working not as expected. We contribute to this challenge and present a framework for integrating different simulation models and diagnoses. The framework utilizes combining functional mock-up units (FMUs) that may originate from modeling environments like OpenModelica[1] with ordinary programming languages like Java or Python. We use these language capabilities to integrate diagnosis functionality. The architecture of our framework is based on the client-server pattern and implemented using Docker containers.

Using our framework, we can easily add diagnoses into systems. In addition, we can use this framework for carrying out verification and validation of the system functionality enhanced with diagnosis capabilities. In this manuscript, we present the framework and show the integration of diagnosis. For the latter purpose, we make use of a simple example. We will make the framework and the underlying diagnosis engine available for free and as open-source. The framework contributes to research area of Edge Artificial

[1] see https://openmodelica.org

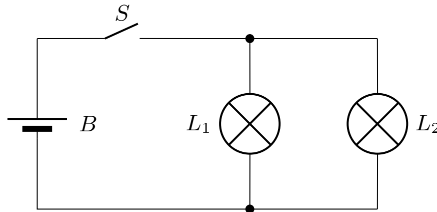


Figure 9.1 A simple electric circuit comprising bulbs, a switch and a battery.

Intelligence because it enables the direct use of diagnosis functionality that is based on Artificial Intelligence methodology in systems without the necessity for communication with other systems.

We structure the paper as follows. First, we discuss the foundations behind the used diagnosis method, i.e., model-based diagnosis. Afterwards, we describe the simulation framework that is based functional mock-up units using a small example. We further show how diagnosis can be integrated into this framework, and finally we conclude the paper.

9.2 Model-based Diagnosis

Diagnosis, i.e., the detection of failures and the identification of faults, have been of interest for several decades. In the early eighties of the last century, Davis and colleagues [1][2] introduced the basic concepts behind model-based diagnosis. The idea is to utilize a model of the system directly for detecting and locating faults. Reiter [5] formalized the idea utilizing first-order logic. For a more recent paper we refer to Wotawa and Kaufmann [8] where the authors introduced how advanced reasoning systems can be used for computing diagnosis. For recent applications of diagnosis in the context of CPS have a look at [3][9][7][6].

In the following, we illustrate the basic concepts using a small example circuit comprising a battery B, a switch S, and two bulbs L_1, L_2. The bulbs are put in parallel and both should provide light when the switch is turned on and the battery is not empty. Otherwise, both bulbs do not deliver any light. We depict the circuit in Figure 9.1. If we know that the switch S is on, and the battery is working as expected, then we also would expect both bulbs to be illuminated. In case one bulb is emitting light but the other is not, we would immediately to derive that the bulb with no transmitting light is broken.

To compute diagnoses from system models, we first need to come up with a model of the system that we want to diagnose. Such models comprise components and their connections, via ports. Hence, in the following, we discuss the component models, and a model of connections separately. For the electric circuit, we simplify modelling by only considering that components like batteries are providing electrical power, some are transferring power like switches, and others are consuming power. Furthermore, we utilize first-order logic for formalization where we follow Prolog syntax[2]. For all the component models we describe how values are computed assuming that the component is of a particular type and that it is working as expected. For the type we use a predicate $\mathrm{type}\backslash 2$ and for stating the component to be correct a predicate $\mathrm{nab}\backslash 1$.

Battery A component X that is a battery is when working correctly providing a nominal power at its output.

```
val(pow(X),nominal) :- type(X,bat), nab(X).
```

Switch A component X that is a switch works as follows. If it is on and working as expected, then the output must have the same value as the input port and vice versa. If it is off, the switch is not transferring any power.

```
val(out_pow(X),V) :-   type(X,sw), on(X),
                       val(in_pow(X),V), nab(X).
val(in_pow(X),V) :-    type(X,sw), on(X),
                       val(out_pow(X),V), nab(X).
val(out_pow(X),zero) :- type(X,sw), off(X), nab(X).
```

Lamp A lamp X is on, whenever there is a power on its input. If it emits light, then there must be power on its input. If there is no power at the input of X, then the light must be off.

```
val(light(X),on) :- type(X,lamp), val(in_pow(X),
                    nominal), nab(X).
val(in_pow(X), nominal) :-  type(X,lamp),
                            val(light(X),on).
val(light(X),off) :-   type(X, lamp),
                       val(in_pow(X),zero), nab(X).
```

[2]We are using Prolog syntax because recent solvers like Clingo (see https://potassco.org/clingo/) are relying on it.

For completing the model, we introduce connections using a predicate $\text{conn}\backslash 2$ that allows to state two ports to be connected. The behaviour of a component comprises the transfer of values in both directions, and stating that it is impossible to have different values at a connection. The following rules are covering this behaviour:

```
val(X,V) :- conn(X,Y), val(Y,V).
val(Y,V) :- conn(X,Y), val(X,V).

:- val(X,V), val(X,W), not V=W.
```

To use a model for diagnosis we only need to define the structure of the system making use of the component models. For the two bulb example, we define a battery, a switch, and two bulbs that are connected accordingly to Figure 9.1.

```
type(b, bat).
type(s, sw).
type(l1, lamp).
type(l2, lamp).

conn(in_pow(s), pow(b)).
conn(out_pow(s), in_pow(l1)).
conn(out_pow(s), in_pow(l2)).
```

To use this model for diagnosis, we further need observations. We might state that the switch s is on, bulb l1 is not on but l2 is. Again we can make use of Prolog to represent this knowledge as facts:

```
on(s).
val(light(l1),off).
val(light(l2),on).
```

When using a diagnosis engine like described in [8] we obtain one single fault diagnosis $\{l1\}$. But how is this working? The diagnosis engine makes use of a simple mechanism. It searches for a truth setting to the $\text{nab}\backslash 1$ predicates, such that the model together with these assumptions is not leading to a contradiction. When assuming l1 to be not working, the fact that lamp l2 is on can be derived. However, we cannot derive anything else that would lead to a contradiction.

Note that this simple model is also working in other more interesting cases. Let us assume that the switch is on but no light is on. For this case, the diagnosis engine delivers three diagnoses: $\{b\}$, $\{s\}$, and $\{l1, l2\}$ stating the either the battery is empty, the switch is broken, or both lamps are not working at the same time. Another interesting case that might occur is setting the switch to off, put still one lamp, i.e., $l1$ is on. In this case we only obtain a double fault diagnosis $\{s, l2\}$ stating that the switch is not working as expected and lamp $l2$ as well.

9.3 Simulation and Diagnosis Framework

In the following section, we introduce a framework making use of two collaborating tools, comprising a simulation environment for function mock-up unit (FMU)[3] models and a diagnose application based on the theorem solver Clingo[4]. Figure 9.2 gives a brief overview of the framework and the operating principles. The FMU simulation tool server is utilized to run a CPS model within the given simulation environment, whereas the client enables to control the simulation. The separation enables to execute other applications, tools and methods after each simulation time step update, as the ASP Diagnose Tool (see Section 9.3.2). The mentioned tool receives the observations provided by the simulation framework and a settings configuration to compute the diagnose of a system, based on an abstract model, developed with the declarative programming language ASP (Answer Set Programming). Further, the diagnose may be used to control the inputs and parameter to restore a safe operating system or to bring the system in a state to prevent harm to the system or environment.

9.3.1 FMU Simulation Tool

The developed application provides an entire environment to load, configure, run, observe and control simulations related to CPS models. In general, the application is set up as a client-server system to distribute the structure between the provider of a service, the server, and the service requester, the client. The service executed on the server is defined as the simulator environment providing the options to observe and control the simulation by client requests during run-time. The reason of using a client-server

[3]see https://fmi-standard.org
[4]see https://potassco.org/clingo/

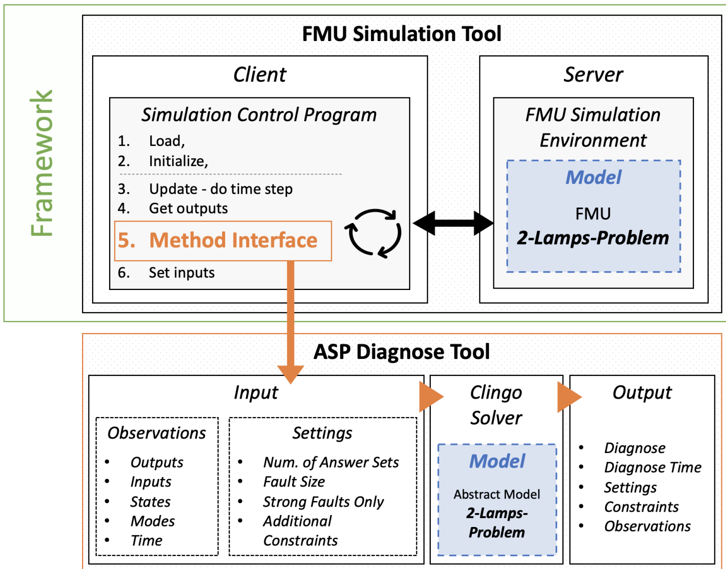


Figure 9.2 Illustration of the simulation and diagnose environment as well as the overall operating principles. The framework of the FMU Simulation Tool provides an interface to enable the integration of a diagnose tool and/or other methods. The models can be substituted by any others in the provided framework.

system is to detach the simulation environment and the observation/control process. The separation enables the user to utilize individual programming environments/languages as client, whereas the server works independent to the selected client environment, receiving and sending the control commands and simulation observations via a REST (Representational State Transfer) application programming interface. In order to run a simulation of a CPS model with the described application, a fundamental requirement is to generate a standardized FMU from the given model. Common modeling software as OpenModelica or Matlab[5] have a FMU generation tool already implemented, but there are also other applications, as e.g. UniFMU [6], which are capable of generating a FMU from different language source code (Python, Java or C/C++). A FMU enables to use a general simulation environment for all kind of models, although they are build on different sources. The simulation environment is developed to execute a step by step

[5] see https://de.mathworks.com/products/matlab.html
[6] see https://github.com/INTO-CPS-Association/unifmu

(for a given time step) simulation. To enable that feature, it is essential that the FMU is generated as a co-simulation model. Within a co-simulation setup, the numerical solver is embedded and supplied by the generated FMU. By the provided interface methods, the FMU can be controlled by setting the inputs and parameter, computing the next simulation time step, and reading the resulting observations. The given setup, enables to execute tools and methods while the simulation is paused after a simulated time step.

9.3.2 ASP Diagnose Tool

To enable diagnoses based on observations of a given CPS model, we developed a diagnose tool. This tool is built up on the theorem solver Clingo and makes use of the provided methods within a Python environment. In addition the tool provides extended functionalities, e.g. including observations as simulation outputs, inputs, states, modes or time and applying optional settings as limiting the number of required answer sets, setting the maximum fault size search space for abnormal component behaviour, considering additional fault modes and adding other constraints to be considered.

The tool is designed to iterate through each fault size in ascending order, whereas fault size zero indicates a normal operating system without detecting any abnormal behaviour in the diagnosed components. The procedure is repeated for each fault size, except when the model is satisfied for fault size zero, what is interpreted as no abnormal component is present for the given observation. The detailed theorem solver implementation structure is shown in algorithm 1, which was initially introduced and applied by Wotawa, Nica and Kaufmann [3]. In the following, we briefly describe the setup of the stated algorithm. First the input model is initiated, defined as an abstract model (M), comprising the system description (SD), observations (Obs) and additional fault modes (FM) to guide the diagnosis search. We start with an empty diagnosis set (DS) and compute diagnosis of a certain size, iterating from 0 to n. Line 4 shows how the limitation of the number for abnormal predicates is applied to the model (M_f), before the solver is called (line 5). A specified answer set is returned and filtered for abnormal predicates (S) only. To prevent the multiple occurrence of abnormal elements (C) in the iterations, the corresponding integrity constraints are added to the model (M_f) as stated in line 12. In relation to the given example in Section 9.2, a integrity constraint at fault size 1 could be stated as `:- ab(l1).` for a detected abnormal behaviour of the component lamp 1.

Besides the main diagnose algorithm, the tool enables different output options to simplify the evaluation of the received diagnose. Thus, the received data can be exported in a JSON file, CSV file or directly printed in the terminal during run-time. The output results are the detailed computed diagnose, the total number of found diagnosis for each fault size, an indicator for strong faults and the diagnose time separated for each fault size and in total. As input, the tool requires the Prolog model, representing the CPS as abstract model (see Section 9.2), and the related observation/constraint file with all necessary input information to execute the diagnose process.

In reference to Figure 9.2, we show the simulation tool update loop, where an update is triggered and the observations are received. Further the observations are passed by the method interface as input to the implemented diagnose tool. Before calling the diagnose, some configurations are specified, as the abstract model, the maximum number of computing answer sets, the maximum fault size of interest and the observations, which are generated based on the simulation output information. In addition, the diagnose output format, e.g., JSON or CSV can be selected. Last, the ASP theorem solver with the given model, configuration and simulation observations is executed. After receiving the diagnose result of the current time frame, it is stored in the defined format structure and the simulation is continued with the next time step in the loop.

9.4 Experiment

To show the applicability of the framework, we make use of the two-lamps-model concept as shown in Figure 9.1. For the simulation, a model of the two-lamps-model (see Listing 1) is generated in OpenModelica comprising a battery ($5.0V$), a closing switch and two light bulbs (100Ω). Besides the connection of each component, the model also describes inputs, which can be set during the simulation. These inputs are covering the fault type of each component and the operational switch logic. To give an example of the component programming, the switch model is shown in more detail at Listing 2. Besides the component mode, the equations also represent the behaviour based on different fault states, e.g. a broken switch, resulting in an infinite high internal resistor value equal to an open electrical circuit. An equivalent fault state is implemented for each component as shown in Table 9.1.

Moreover the OpenModelica model is converted into a co-simulation FMU, which enables to use the model in the described FMU simulation tool.

Algorithm 1 ASPDiag(SD, Obs, FM, n)
For a more detailed description of the algorithm see [3].

Input: An ASP diagnosis model M, and the desired cardinality n
Output: All minimal diagnoses up to n

1: Let DS be $\{\}$
2: Let M_f be M.
3: **for** $i = 0$ **to** n **do**
4: $M'_f = M_f \cup \{$:- not numABs(i). $\}$
5: $S = \mathcal{F}\left(\textbf{ASPSolver}(M'_f)\right)$
6: **if** i is 0 **and** S is $\{\{\}\}$ **then**
7: **return** S
8: **end if**
9: Let DS be $DS \cup S$.
10: **for** Δ in S **do**
11: Let $C = AB(\Delta)$ be the set $\{c_1, \ldots, c_i\}$
12: $M_f = M_f \cup \{$:- ab(c$_1$), ..., ab(c$_i$). $\}$.
13: **end for**
14: **end for**
15: **return** DS

```
model Two_Lamp_Circuit
  PhysicalFaultModeling.PFM_Bulb bulb1(r = 100.0);
  PhysicalFaultModeling.PFM_Bulb bulb2(r = 100.0);
  PhysicalFaultModeling.PFM_Switch sw;
  PhysicalFaultModeling.PFM_Ground gnd;
  PhysicalFaultModeling.PFM_Battery bat(vn = 5.0);
equation
  connect(gnd.p, bat.m);
  connect(bat.p, sw.p);
  connect(sw.m, bulb1.p);
  connect(sw.m, bulb2.p);
  connect(bulb1.m, gnd.p);
  connect(bulb2.m, gnd.p);
end Two_Lamp_Circuit;

model Two_Lamp_Circuit_Testbench
  PhysicalFaultModeling.Two_Lamp_Circuit sut;
  input FaultType bat_state(start=FaultType.ok);
  input OperationalMode switch_mode(start=OperationalMode.close);
  input FaultType switch_state(start=FaultType.ok);
  input FaultType bulb1_state(start=FaultType.ok);
  input FaultType bulb2_state(start=FaultType.ok);
equation
  sut.sw.mode = switch_mode;
  sut.bat.state = bat_state;
  sut.sw.state = switch_state;
  sut.bulb1.state = bulb1_state;
  sut.bulb2.state = bulb2_state;
end Two_Lamp_Circuit_Testbench;
```

Listing 1 OpenModelica model of a two-lamp electrical circuit with fault injection capability to each used component. The component connections are specified to describe the same electrical circuit as given in Figure 9.1.

```
model PFM_Switch
  extends PhysicalFaultModeling.PFM_Component;
  PhysicalFaultModeling.OperationalMode mode(start=OperationalMode.open);
  Modelica.Units.SI.Resistance r_int;
equation
  v = r_int * i;
  if state == FaultType.ok then
    if mode == OperationalMode.open then
      r_int = 1e9;
    else
      r_int = 1e-9;
    end if;
  elseif state == FaultType.broken then
    r_int = 1e9;
  else
    r_int = 1e-9;
  end if;
end PFM_Switch;
```

Listing 2 OpenModelica model of a switch component including a mode `{open, close}` and fault state `{ok, broken, short}` implementation logic.

Table 9.1 CPS Model component state description for the light bulb, switch and battery. All used states, including fault states of the components are shown.

Component	*State*	*Description*
light bulb (`bulb`), switch (`sw`)	`ok`	ordinary behaviour
	`broken`	open connection in eletrical circuit
	`short`	short in the electrical circuit
battery (`bat`)	`ok`	ordinary behaviour
	`empty`	empty battery fault

In order to simulate the model behaviour in detail, the update time step is set to 0.01 seconds. In addition, the fault injection during run-time is configured to trigger a single light bulb fault at 0.2 seconds and a switch fault after 0.3 seconds, which is described in detail at the simulation part of Figure 9.4.

For the diagnose part, we make use of the described abstract model of the electrical two-lamps circuit (see Section 9.2). The overall framework is built up in a way, that a diagnose is computed after each simulated time step and is based on the actual observations (simulation outputs, parameter and inputs). The use of a co-simulation FMU, allows a step-by-step simulation, which enables to pause the simulation during the diagnose process and continuing afterwards. Therefore, the diagnose time effort has no impact on the overall simulation results.

Figure 9.3 shows the observed signals for the current flow in the battery, light bulb 1 and 2 as well as the actual switch mode. Further the injected faults are highlighted at the correlated time point. In Figure 9.4 a table represents the observations for the three interesting time sections, as the normal behaviour, a broken light bulb and a broken switch. After reaching simulation time 0.05

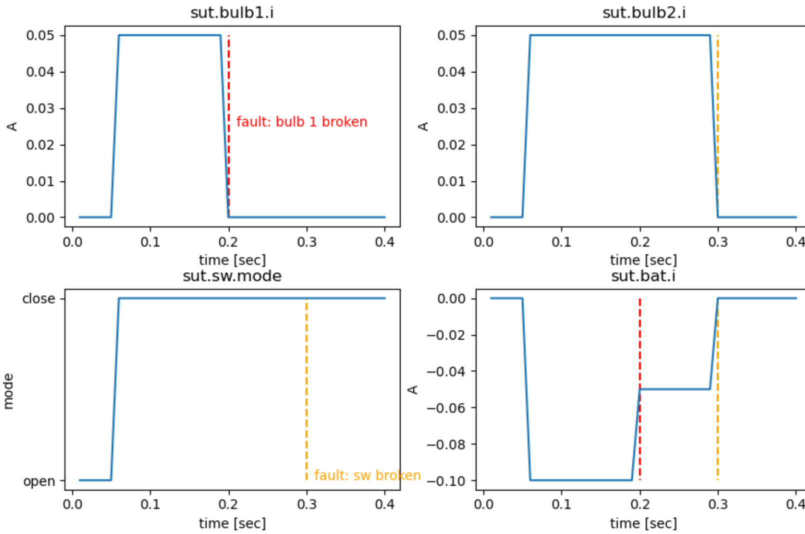


Figure 9.3 Simulation showing the measured signal output of the two bulbs, switch and the battery. For this example a fault injection (`broken`) in bulb 1 after 0.2 seconds (red indicator) and a fault injection (`broken`) to the switch after 0.3 seconds (orange indicator) is initiated.

seconds, the switch mode is changed from `open` to `closed` and the model shows the expected ordinary behaviour without any abnormal components. Both light bulbs are operating at an expected current consumption of 0.05 A. These observations are translated to a readable input format for the diagnose tool, which is shown in the corresponding status row "Observation" (see Figure 9.4). In regards to the abstract model and the observation input, the diagnose tool computed a satisfied model at fault size zero, which concludes an expected ordinary behaviour of all considered components.

The time section at 0.2 seconds shows the behaviour with a broken light bulb. Thus the current consumption of bulb 1 immediately drops to 0.0 A and the diagnose observation changes from mode `on` to `off`. Since the main power switch is still closed and bulb 2 is in active mode `on`, the diagnose model concludes component bulb 1 as abnormal `ab(l1)`. The next investigated fault (`broken`) is injected to the closed switch. Since the power supply for both light bulbs is not given, the current consumption drops to 0.0 A. The diagnose model concludes as expected an abnormal switch (`ab(s)`) or battery (`ab(b)`) based on the given observations for single faults. Under consideration of double faults, the computed diagnose

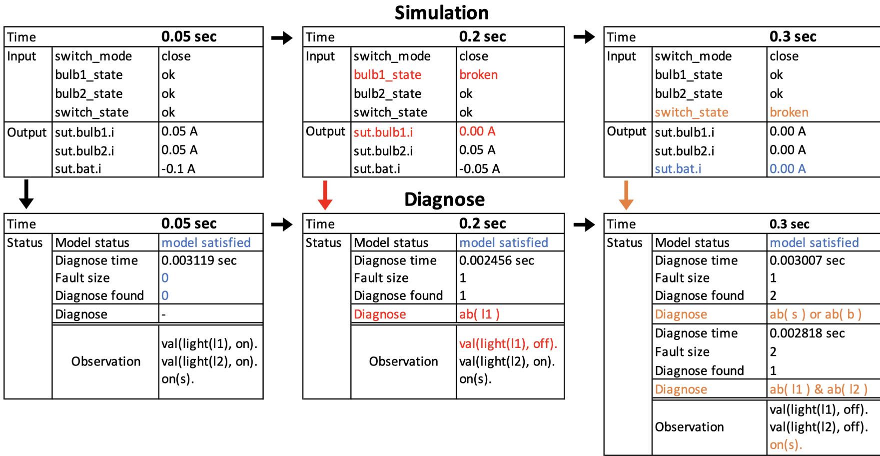


Figure 9.4 Simulation and diagnose output results based on the electrical two-lamps circuit with a broken bulb after 0.2 seconds and a broken switch at 0.3 seconds. The upper tables illustrate the simulation input/output signals, which are used as observation for the diagnose (lower tables) part. Based on the given observations for the three selected time steps, different diagnose results are obtained.

shows a combination of an abnormal behaviour of light bulb 1 and bulb 2 (`{ab(l1), ab(l2)}`), which is also a possible solution for the given observation.

9.5 Conclusion

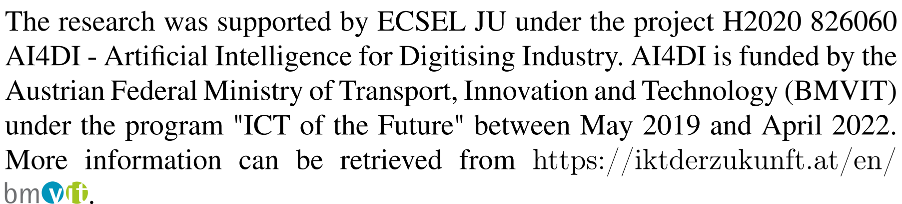
In this paper, we have shown how to use an automated diagnosis method within a simulation framework for a CPS (cyber-physical system). For this purpose we introduced the foundations behind the model-based diagnosis method based on a simple electric circuit model comprising two light bulbs, a switch and battery. Next we describe a framework for simulating the developed CPS model with the ability of fault injection during run-time. In order to run the model in the given framework, it is essential to generate a functional mock-up unit (FMU) based on the developed electrical two lamp circuit model. By providing the FMU in co-simulation configuration, the simulation can run in a step-by-step mode (time steps), which enables to call other functions, as for example the diagnose method, while the simulation is paused and continued with the next time step.

Besides the physical electrical circuit model, an abstract model for diagnosis is developed in the declarative programming language Prolog. For computing the diagnose based on observations of the model simulation, we introduce a tool which uses the theorem solver Clingo and offers additional productive options. The tool is developed to automate the process of searching for abnormal components at each fault size (in ascending order). To prevent multiple occurrence of abnormal components in higher fault sizes, the derived results are continuously added as constraints to the model.

In order to demonstrate the concept of the simulation framework with the automated diagnose tool, we executed an experiment based on the described electrical two-lamps circuit model with the capability of fault injection to the light bulbs and switch. After each time step of simulation, the received observations are forwarded as input to the diagnose tool. The diagnose tool enables to detect the injected faults in a fast and accurate way, as a single bulb fault or even the more interesting case, when a switch erroneously indicates a closed position although both light bulbs are powered off. In this case, we obtain a single fault for an abnormal switch or battery behaviour, and a double fault stating an abnormal behaviour for both light bulbs in combination.

For the purpose of deploying the diagnose tool on a system applied under real environmental conditions, validation and verification is a fundamental process. Thus we make use of a simulated environment framework, enabling a high test case coverage of scenarios with abnormal component behaviour of the system under test. In addition, the required time to conclude a diagnose, may also lead to issues and need to be considered in the evaluation. Future research includes investigating more complex CPSs by making use of the discussed simulation framework in combination with the diagnose tool and further development of both tools.

Acknowledgments

The research was supported by ECSEL JU under the project H2020 826060 AI4DI - Artificial Intelligence for Digitising Industry. AI4DI is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between May 2019 and April 2022. More information can be retrieved from https://iktderzukunft.at/en/


.

References

[1] R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, M. Shirley, and S. Polit. Diagnosis based on structure and function. In *Proceedings AAAI*, pages 137–142, Pittsburgh, August 1982. AAAI Press.

[2] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[3] D. Kaufmann, I. Nica, and F. Wotawa. Intelligent agents diagnostics - enhancing cyber-physical systems with self-diagnostic capabilities. *Adv. Intell. Syst.*, 3(5):2000218, 2021.

[4] N. Muscettola, P. Pandurang Nayak, B. Pell, and B. C. Williams. Remote agent: to boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1):5–47, 1998. Artificial Intelligence 40 years later.

[5] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[6] F. Wotawa. Reasoning from first principles for self-adaptive and autonomous systems. In E. Lughofer and M. Sayed-Mouchaweh, editors, *Predictive Maintenance in Dynamic Systems – Advanced Methods, Decision Support Tools and Real-World Applications*. Springer, 2019.

[7] F. Wotawa. Using model-based reasoning for self-adaptive control of smart battery systems. In Moamar Sayed-Mouchaweh, editor, *Artificial Intelligence Techniques for a Scalable Energy Transition – Advanced Methods, Digital Technologies, Decision Support Tools, and Applications*. Springer, 2020.

[8] F. Wotawa and D. Kaufmann. Model-based reasoning using answer set programming. *Applied Intelligence*, 2022.

[9] F. Wotawa, O. A. Tazl, and D. Kaufmann. Automated diagnosis of cyber-physical systems. In *IEA/AIE (2)*, volume 12799 of *Lecture Notes in Computer Science*, pages 441–452. Springer, 2021.