14

On the Verification of Diagnosis Models

Franz Wotawa and Oliver Tazl

Graz University of Technology, Austria

Abstract

Enhancing systems with advanced diagnostic capabilities for detecting, locating, and compensating faults during operation increases autonomy and reliability. To assure that the diagnosis-enhanced system really has improved reliability, we need – besides other means – to check the correctness of the diagnosis functionality. In this paper, we contribute to this challenge and discuss the application of testing to the case of model-based diagnosis, where we focus on testing the system models used for fault detection and localization. We present a simple use case and provide a step-by-step discussion on introducing testing, its capabilities, and arising issues. We come up with several challenges that we should tackle in future research.

Keywords: model-based diagnosis, testing, verification and validation.

14.1 Introduction

Every system comprising hardware faces the problem of degradation under operation, which impacts its behavior over time. To prevent unwanted behavior that may lead to harm, we have to carry out regular maintenance tasks. Maintenance includes preventive activities like changing the tires of cars when their surfaces do not meet regulations anymore and looking at errors occurring during operation. The latter requires root cause identification, i.e., searching for components we have to repair for failure recovery. There is no doubt that the maintenance and diagnosis of engineered systems are of practical importance and, therefore, worth being considered in research.

If we aim to support maintenance personnel carrying out diagnoses, we need to automate the fault detection and localization activities. Since the beginning of artificial intelligence, diagnosis has been an active research field leading to expert systems and later to model-based diagnosis. The idea behind model-based diagnosis is to use system models for localizing the root causes of detected failures. Early work includes Davis and colleagues [3] papers discussing the basic ideas and concepts behind model-based reasoning. Later, Reiter [15] formalized the idea utilizing first-order logic. Based on these foundations, several authors have discussed several applications of model-based reasoning for solving real-world problems. Applications range from power supply networks [1], the automotive domain [13], space probes [14], robotics [7], self-adaptive systems [16], to debugging [6]. For a more recent paper, we refer to Wotawa and Kaufmann [22], where the authors introduced how advanced reasoning systems can be used for computing diagnosis. For recent applications of diagnosis in the context of cyber-physical systems, have a look at [9, 23, 21, 20].

In the following, we illustrate the basic ideas and concepts of model-based reasoning using a small example circuit comprising a battery B, a switch S, and two bulbs L_1, L_2. We depict the circuit in Figure 14.1. If we switch on S, we expect both bulbs to transmit light when we assume the correctness of every component. It is important to consider such correctness assumptions. For example, if we switch on S, and only one bulb (e.g., L_1) is on, and the other (e.g., L_2) is not, we conclude a broken bulb. But how can we do this? We may consider a model for each component, e.g., a correct battery provides electricity, a switch in the on state takes the electricity from the battery and transmits it to the bulbs, and a correct bulb produces light if there is electricity available. When we assume that all components are working, we receive a contradiction from this model. This is due to bulb L_2 that should produce light but we do not observe it. If we assume all components except L_2 to be correct, there is no contradiction anymore, and we have identified the root cause, i.e., L_2.

A prerequisite of model-based diagnosis is the availability of a system model (or model in short). Modeling is not a trivial task. For model-based diagnosis, we need models formulated in a language that a reasoning system can use for deriving logical conclusions. Models are abstract representations of the system structure and behavior. Only parts of the system classified as components in the model can be part of a derived root cause. Wires or connectors need to be stated as components if we want to have them included in a diagnosis. In model-based diagnosis, only components used in models

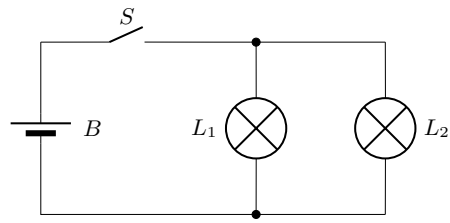


Figure 14.1 A simple electric circuit comprising bulbs, a switch and a battery.

can be part of a root cause. It is also worth noting that we can use uncertainty in model-based diagnosis. De Kleer and Williams [4] formalized the use of fault probabilities of components for searching for the most probable diagnosis. In addition, de Kleer and Williams introduced an algorithm for selecting the optimal probing locations for minimizing probing steps for identifying a single diagnosis.

In this manuscript, we do not focus on the diagnosis methods and processes themselves. Instead, we provide a discussion on how to verify diagnosis models. The challenge of model verification is of uttermost importance for assuring that systems equipped with diagnosis functionality work correctly. Although we may use some of the presented results for verifying diagnosis models generated by machine learning, we consider models for model-based reasoning in the context of this paper. For testing machine learning, we refer the interested reader to a recent survey [24].

The challenge of model-based diagnosis and other logic-based reasoning systems is not that novel. Wotawa [17] introduced the use of combinatorial testing and fault injection for testing self-adaptive systems based on models. The same author also discussed the use of combinatorial testing and metamorphic testing for theorem provers in [18] and the general challenge [19]. In any of these papers, the focus is on testing the implementation and not the underlying models. Koroglu and Wotawa [10] also contributed to the challenge of verifying the reasoning system but focused on the underlying compiler that allows reading in logic theories, i.e., system models. Hence, testing the system models used for diagnosis is still an open challenge worth tackling for quality assurance.

We organize this paper as follows: In Section 14.2, we introduce the testing challenge in detail including a first solution. Afterward, we present the results when using the provided solution in a small case study. Finally, we discuss open issues, and further challenges, and conclude the paper.

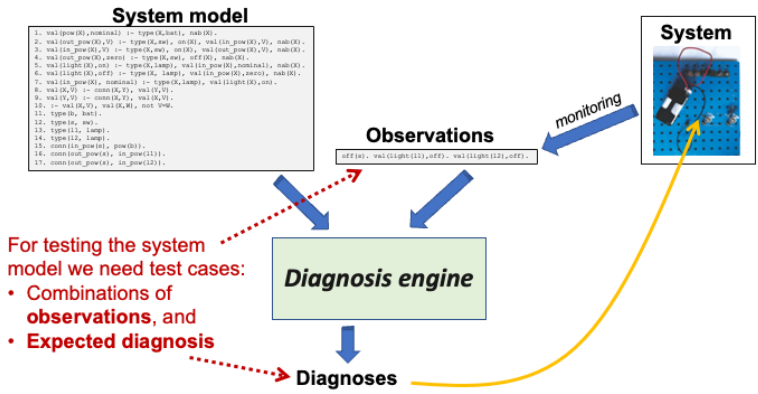


Figure 14.2 The model-based diagnosis principle and information needed for testing.

14.2 The Model Testing Challenge

Before discussing the model testing challenge in detail, we briefly summarize model-based diagnosis and the required information. In Figure 14.2 we depict the basic architecture behind every model-based diagnosis system. On the right side, we have a (physical) system from which we extract observations. On the upper left side, we have a model of the system. This model shall represent the system in a way such that expected observations can be derived. The model and the observations are passed to a diagnosis engine, which tries to find an arrangement of health states to components such that no contradiction can be derived. In the simplest case, we only know the correct behavior of components. We use a logic predicate `nab\1` to represent the corresponding health state. The diagnosis engine itself is assumed to be based on either a theorem prover or a constraint solver. It delivers a set of diagnoses. Each diagnosis itself is a set of faulty components. If the set of diagnoses comprises the empty set, we know that all components are working as expected.

It is worth noting that in the context of this paper, we are not interested in outlining the details regarding model-based diagnosis, the modeling principles, and algorithms. We solely focus on testing, and specifically on testing the system model. What we can take with us from Figure 14.2 are the inputs and outputs to the diagnosis engine comprising the model, the observations, and the computed diagnoses. If we want to verify the implementation of

the diagnosis engine, we can use models and observations together with the corresponding expected diagnoses to define a test case. However, when we want to test the models, which are usually divided into two parts, the component models, and the structure of the system, we have to further think about underlying assumptions and prerequisites.

First, we have to assume that the diagnosis engine itself is correct. This means that the diagnosis engine is delivering the right diagnoses for a given model and observations. Testing the implementation of the diagnosis engine might also comprise testing the underlying theorem prover or constraint solver, the implementation of the diagnosis algorithm, and the compiler that is used to load a model and the observations into the diagnosis engine.

Second, the observations themselves describe the data that have been observed from the system. Usually, we do not use the raw data obtained from the system directly. The data is usually mapped to logical representations. Because we are only focusing on the verification of models used for diagnosis, there might also be faults occurring that originate from the mapping of data to their logical representations. For verifying the model, we do not need to deal with this topic. We can stay with the abstract representation of real observations for testing.

Finally, we assume that models can be divided into component models and structural models. We further assume that the component models are generally valid and can be used in several systems. This assumption is of particular importance because one argument in favor of model-based diagnosis is its flexibility in adapting to different systems and its model re-use capabilities.

Let us now come up with a definition of the challenge of testing diagnosis models where we have the following information given:

1. A model M for components of given types and their connections.

For testing we want to have the following:

1. A set of systems Σ and for each system $S \in \Sigma$ a model M_S representing the structure, i.e., its components and connections.
2. For each system S, we want to have a set of inputs, i.e., possible observations, and a set of expected diagnoses. Note that observations include inputs and outputs of a system, and control commands (like opening or closing a switch).

Note that the systems, as well as their inputs, must be obtained such that they may lead the diagnosis engine to compute different values. This principle

is well-known in testing where testers focus on revealing faults and try to bring an implementation into a state of failure. For stating the problem, we do not rely on automation. Test cases for diagnosis models, and in particular the behavioral part, may be developed manually or using any method for automated test case generation (if possible).

In practice, we might be interested in testing a particular model comprising a structural and behavioral part of a given system. For this variant of the general model testing challenge, we only need to come up with observations and expected diagnoses. In the next section, we discuss generating test cases using the two-bulb example as a use case.

14.3 Use Case

In this section, we use the two-bulb example from Figure 14.1 as a use case for diagnosis model testing. We developed the diagnosis model using the input format of the Clingo[1] theorem prover that relies on the logic programming language Prolog. In Figure 14.3 we see the source code of the model. In Line 1, the ordinary behavior of a battery is given. In case the battery is correctly working (and the predicate `nab\1` is true), the battery provides a nominal output at the `pow` port. In lines 2-4, we formalize the model of a switch. A switch is transferring the power from the `in_pow` to the `out_pow` port and vice versa if it is correctly working an `on`. If the switch is `off`, there is no power at the output. Similarly, in lines 5-7, we see the behavior model of bulbs. If there is nominal power on the input, and the bulb is working fine, then the bulb is shining. If there is no power, there is also no light. If there is a light, we know that there must be electricity provided.

In lines 8-10, we have the connection model, stating that there is a transfer from one port of a component to another, and their values must be the same. The latter is stated in Line 10. Afterward, we have the structural model of the circuit. First, we define the components of the circuit `b`, `s`, `l1`, `l2` for the battery, switch, lamp 1 and lamp 2 respectively. Second, we state the connections between the ports of the components.

For testing the model of the particular two-bulb system, we have to provide test cases comprising observations (which work as the inputs to the model) and the expected diagnoses (which are the expected outputs). For the two bulb example, the position of the switch (`on`, `off`), and the state of the

[1] see `https://potassco.org`

```
1.  val(pow(X),nominal) :- type(X,bat), nab(X).
2.  val(out_pow(X),V) :-    type(X,sw), on(X),
                            val(in_pow(X),V), nab(X).
3.  val(in_pow(X),V) :-     type(X,sw), on(X),
                            val(out_pow(X),V), nab(X).
4.  val(out_pow(X),zero) :- type(X,sw), off(X), nab(X).
5.  val(light(X),on) :- type(X,lamp),
                            val(in_pow(X),nominal), nab(X).
6.  val(light(X),off) :-    type(X, lamp),
                            val(in_pow(X),zero), nab(X).
7.  val(in_pow(X), nominal) :-  type(X,lamp),
                                val(light(X),on).
8.  val(X,V) :- conn(X,Y), val(Y,V).
9.  val(Y,V) :- conn(X,Y), val(X,V).
10. :- val(X,V), val(X,W), not V=W.
11. type(b, bat).
12. type(s, sw).
13. type(l1, lamp).
14. type(l2, lamp).
15. conn(in_pow(s), pow(b)).
16. conn(out_pow(s), in_pow(l1)).
17. conn(out_pow(s), in_pow(l2)).
```

Figure 14.3 A model for diagnosis of the two lamp example from Figure 14.1 comprising the behavior of the components (lines 1-7) and connections (lines 8-10), and the structure of the circuit (lines 11-18).

two bulbs regarding light emission (`on`, `off`) serve as the inputs. It is worth noting that the power supply of the battery might also be observed. However, for the initial testing, we only consider those observations where we do not require additional equipment for measurement in practice. Nevertheless, for testing, we may also consider more observations.

When having 3 observations each having a domain comprising 2 values, we finally obtain 8 test cases covering all combinations. We depict this test cases in Table 14.1. Note that the first two test cases (which are highlighted in gray) cover the correct behavior of the system, where the switch is used to turn on and off lamps. Therefore, we see the empty set as the expected diagnosis in the corresponding column. The other test cases formalize an incorrect behavior of the two-bulb circuit.

For testing the model, we run our diagnosis engine `model_diagnose` using the observations of a test case. In Clingo adding observations to models can be simple done via linking the model into a file where we state the

Table 14.1 All eight test cases used to verify the 2-bulb example comprising the used observations and the expected diagnoses. The **P/F** column indicates whether the original model passes ($\surd$) or fails ($\times$) the test.

	Observations	**Expected diagnoses**	**P/F**
1	`on(s). val(light(l1),on). val(light(l2,on)).`	$\{\{\}\}$	$\surd$
2	`off(s). val(light(l1),off). val(light(l2,off)).`	$\{\{\}\}$	$\surd$
3	`off(s). val(light(l1),on). val(light(l2,off)).`	$\{\{s, l2\}\}$	$\surd$
4	`off(s). val(light(l1),off). val(light(l2,on)).`	$\{\{s, l1\}\}$	$\surd$
5	`off(s). val(light(l1),on). val(light(l2,on)).`	$\{\{s\}\}$	$\surd$
6	`on(s). val(light(l1),on). val(light(l2,off)).`	$\{\{l2\}\}$	$\surd$
7	`on(s). val(light(l1),off). val(light(l2,on)).`	$\{\{l1\}\}$	$\surd$
8	`on(s). val(light(l1),off). val(light(l2,off)).`	$\{\{b\}, \{s\}, \{l1, l2\}\}$	$\surd$

observations. For the first test case the file `tle_obs1.pl` comprises the following statements:

```
#include ẗwo_lamps_example.pl.̈
on(s).
val(light(l1),on).
val(light(l2),on).
```

The first line includes the model we show in Figure 14.3, which we store in the file `two_lamps_example.pl`. For executing a test case, we run the diagnosis engine in a shell using the following command: `./model_diagnose -f tle_obs1.pl -fault 2`. In this call, we ask for diagnoses comprising up to two components, which we do via setting the parameter `-fault` to `2`. Finally, we used a shell script to carry out all test cases. We see the outcome of testing in column `P/F` in Table 14.1. The model passes all tests successfully.

After checking the correctness of diagnosis results obtained when using the model, we wanted to evaluate the quality of the test suite. In software engineering, measures like code coverage or the mutation score are used for this purpose. Estimating code coverage, i.e., the number of rules used to derive a contradiction for diagnosis is difficult because theorem provers

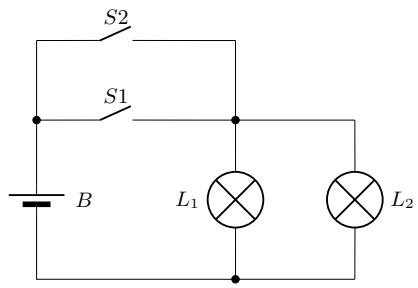
Table 14.2 Running 7 model mutations Mi, where we removed line i in the original model of Figure 14.3, using the 8 test cases from Table 14.1.

	M1	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**
1	√	√	√	√	√	√	√
2	√	√	√	√	√	√	√
3	√	√	√	×	×	√	×
4	√	√	√	×	×	√	×
5	√	√	√	×	√	√	×
6	√	√	√	√	×	√	×
7	√	√	√	√	×	√	×
8	×	×	√	√	×	√	√

usually do not provide this information. Therefore, we focused on mutation testing [2, 12]. The underlying idea is to modify a program and to have a look at whether this modification can be detected by the test suite. The mutation score is defined as the fraction of the detected and all mutations. There are some issues when computing the mutation score, for example, equivalent mutants, i.e., changes of the program that are not changing the behavior.

For languages like Java, there are tools, e.g., [8]. In our case, because of a lack of tools, we only removed rules as modification operators. In particular, we were interested in looking at the consequences to the diagnosis results when removing a rule from a component model. We define a mutant Mi as the original program (from Figure 14.3) where we removed the rule in Line i. In Table 14.2 we find the results obtained for each mutant. We see that there are two mutants M3 and M6 that cannot be detected by any test cases. Hence, the mutation score for our test suite is $\frac{5}{7} = 0.7143$. To clarify the reason behind not having a mutation score of 1.0 we analyzed the corresponding rules of mutant M3 and M6. M3 allows transferring electricity also from the output to the input, which might be appropriate when dealing with other circuits. M6 covers the case where there is zero power on the input. Because there are no other rules allowing to derive zero power, this rule does not provide anything for the reasoning process for this use case and can be removed. Please note that the rule might be introduced again when considering a different use case where we have to deal with zero power at the input.

The question that remains is whether the component models can be used for other systems as well. To verify the corresponding property, i.e., the component models are generally applicable, we have to come up with new systems and apply test case generation again. In this use case, we slightly modified the original two-bulb example. We added another switch in parallel such that both provide the functionality of an or-gate. The lamps have to be




```
type(b, bat).
type(s1, sw).
type(s2, sw).
type(l1, lamp).
type(l2, lamp).

conn(in_pow(s1), pow(b)).
conn(out_pow(s1), in_pow(l1)).
conn(out_pow(s1), in_pow(l2)).
conn(in_pow(s2), pow(b)).
conn(out_pow(s2), in_pow(l1)).
conn(out_pow(s2), in_pow(l2)).
```

Figure 14.4 Another simple electric circuit comprising bulbs, switches and a battery. This circuit is an extended version of the circuit from Figure 14.1. On the right, we have the structural model of this circuit in Prolog notation.

off only if both switches are open, i.e., in their off state. See Figure 14.4 for the schematics of the extended two-bulb circuit.

For testing the extended two-bulb circuit, we have to introduce test cases. Similar to the original circuit, we use all combinations of input values, and manually computed the expected diagnoses. We depict the whole test suite in Table 14.3. There we also see the obtained results after automating the test execution using shell scripts. For many test cases, the computed diagnoses are not equivalent to the expected ones. We conclude that the provided model is not generally applicable.

After carefully analyzing the root cause behind this divergence, we identified the rule in Line 4 of the component model (from Figure 14.3) as problematic. This rule states that an open switch assures that there is no power on the output of the switch. Unfortunately, there might be electricity available because of another power supplying component like given in the extended two-bulb example. Unfortunately, we are also not able to remove this rule because otherwise, the behavior of the original two-bulb example would change (see Table 14.2). A solution would be to introduce a specific or-component that takes the outputs of the two switches as inputs and provides power whenever at least one power output has a nominal value.

14.4 Open Issues and Challenges

We can identify the following results and issues from the use case discussed in the previous section.

Table 14.3 Test cases for the extended two-bulb example from Figure 14.4 and their test execution results. In gray we indicate tests that check the expected (fault-free) behavior of the circuit.

	Observations	Expected diagnoses	P/F
1	`on(s1). on(s2).` `val(light(l1,on)). val(light(l2),on).`	$\{\{\}\}$	$\surd$
2	`off(s1). on(s2).` `val(light(l1,on)). val(light(l2),on).`	$\{\{\}\}$	$\times$
3	`on(s1). off(s2).` `val(light(l1,on)). val(light(l2),on).`	$\{\{\}\}$	$\times$
4	`off(s1). off(s2).` `val(light(l1,off)). val(light(l2),off).`	$\{\{\}\}$	$\surd$
5	`on(s1). on(s2).` `val(light(l1,off)). val(light(l2),on).`	$\{\{l1\}\}$	$\surd$
6	`on(s1). on(s2).` `val(light(l1,on)). val(light(l2),off).`	$\{\{l2\}\}$	$\surd$
7	`on(s1). on(s2).` `val(light(l1,off)). val(light(l2),off).`	$\{\{b\},\{s1,s2\}\{l1,l2\}\}$	$\surd$
8	`on(s1). off(s2).` `val(light(l1,off)). val(light(l2),on).`	$\{\{l1\}\}$	$\times$
9	`on(s1). off(s2).` `val(light(l1,on)). val(light(l2),off).`	$\{\{l2\}\}$	$\times$
10	`on(s1). off(s2).` `val(light(l1,off)). val(light(l2),off).`	$\{\{b\},\{s1\}\{l1,l2\}\}$	$\times$
11	`off(s1). on(s2).` `val(light(l1,off)). val(light(l2),on).`	$\{\{l1\}\}$	$\times$
12	`off(s1). on(s2).` `val(light(l1,on)). val(light(l2),off).`	$\{\{l2\}\}$	$\times$
13	`off(s1). on(s2).` `val(light(l1,off)). val(light(l2),off).`	$\{\{b\},\{s2\},\{l1,l2\}\}$	$\times$
14	`off(s1). off(s2).` `val(light(l1,on)). val(light(l2),off).`	$\{\{s1,s2,l2\}\}$	$\surd$
15	`off(s1). off(s2).` `val(light(l1,off)). val(light(l2),on).`	$\{\{s1,s2,l1\}\}$	$\surd$
16	`off(s1). off(s2).` `val(light(l1,on)). val(light(l2),on).`	$\{\{s1,s2\}\}$	$\surd$

- *Testing a model* for a particular system that is based on component models and a structural part *is possible* but requires to identify (i) the input, i.e., observations given to the system, and (ii) the expected diagnosis. From this result the following issues arise:
  - We have to identify the observations given to the system. This might not be an obvious task requiring to analyse the functionality of the system. We may start with observations of the input and the output of the system. But this might not be a complete test suite when considering the mutation score.

  - Furthermore, we have to consider different observations. We may make use of all combinations as we did in the case study. However, for a larger system, this is infeasible, and other approaches are required. Combinatorial testing [11] might be a good starting point for future research.
  - The expected diagnoses have to be computed manually. This is a time-consuming task. Hence, any means for automating this step would be highly appreciated.

- The *generated test suite may not* lead to one that allows for *detecting all faults*. Fault detection capabilities are usually measured using the mutation score. From the use case discussed in the previous section, we see that the mutation score, even when considering only one mutation operator, might be less than 1.0. Related issues and future research activities include:
  - We need to come up with a well-founded theory of mutation testing for logic rules. This also includes considering more mutation operators.
  - There is a need for generating test cases for diagnosis models automatically such that the mutation score can be maximized.
- *Testing should be extended* to check whether the component *models can be used in other systems* as well. What is missing in this context is:
  - The automated generation of different but still relevant systems for practical applications is an open research question. For each of the generated systems, we need to compute test suites and check the correctness of the computed diagnosis. Note that in principle, we have an infinite number of such systems. We have to think about when to stop testing.
  - In case of deviations between the expected diagnoses and the computed ones, someone is interested in identifying the reasons behind them. Hence, we need debugging functionality that may be similar to previous work on debugging knowledge bases [5].

In summary, the main challenge relies on the automation of test case generation. Test cases or at least the expected diagnoses have to be generated manually. Moreover, we need to adapt existing testing methods and techniques for logic representations. Partially there is related work someone can start with. But when compared to corresponding work for ordinary programming languages, available knowledge can be considered minor.

14.5 Conclusion

In this paper, we discussed the use of testing for model-based diagnosis. We focused on assuring the quality of system models used for fault detection and localization. We discussed how to test models and identified arising shortcomings, and future research directions. Testing a system model comes in two flavors: (i) testing a model of a particular system and (ii) testing component models used in different system models. For both, we need to define test cases comprising observations and expected diagnoses. For testing component models, in addition, we need to come up with different systems. Issues and challenges include providing means for answering the question of when to stop testing, giving quality guarantees, and the automation of test case generation.

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/` bmvit.

References

[1] A. Beschta, O. Dressler, H. Freitag, M. Montag, and P. Struss. A model-based approach to fault localization in power transmission networks. *Intelligent Systems Engineering*, 1992.

[2] T. Budd, R. DeMillo, R. Lipton, and F. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. Seventh ACM Symp. on Princ. of Prog. Lang. (POPL)*. ACM, January 1980.

[3] 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.

[4] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[5] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency based diagnosis of configuration knowledge bases. In *Proceedings of the*

European Conference on Artificial Intelligence (ECAI), Berlin, August 2000.

[6] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[7] M. W. Hofbaur, J. Köb, G. Steinbauer, and F. Wotawa. Improving robustness of mobile robots using model-based reasoning. *J. Intell. Robotic Syst.*, 48(1):37–54, 2007.

[8] R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, San Jose, CA, USA, 2014.

[9] 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.

[10] Y. Koroglu and F. Wotawa. Fully automated compiler testing of a reasoning engine via mutated grammar fuzzing. In *In Proc. of the 14th IEEE/ACM International Workshop on Automation of Software Test (AST)*, Montreal, Canada, 27th May 2019.

[11] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. Taylor & Francis, 2013.

[12] J. A. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[13] C. Picardi, R. Bray, F. Cascio, L. Console, P. Dague, O. Dressler, D. Millet, B. Rehfus, P. Struss, and C. Vallée. Idd: Integrating diagnosis in the design of automotive systems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 628–632, Lyon, France, 2002. IOS Press.

[14] K. Rajan, D. Bernard, G. Dorais, E. Gamble, B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. Nayak, N. Rouquette, B. Smith, W. Taylor, and Y.-w. Tung. Remote Agent: An Autonomous Control System for the New Millennium. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, August 2000.

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

[16] G. Steinbauer and F. Wotawa. Model-based reasoning for self-adaptive systems - theory and practice. In *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 187–213. Springer, Switzerland, 2013.

[17] F. Wotawa. Testing self-adaptive systems using fault injection and combinatorial testing. In *Proceedings of the Intl. Workshop on Verification and Validation of Adaptive Systems (VVASS 2016)*, pages 305–310, Vienna, Austria, 2016. IEEE.

[18] F. Wotawa. Combining combinatorial testing and metamorphic testing for testing a logic-based non-monotonic reasoning system. In *In Proceedings of the 7th International Workshop on Combinatorial Testing (IWCT) / ICST 2018*, April 13th 2018.

[19] F. Wotawa. On the automation of testing a logic-based diagnosis system. In *In Proceedings of the 13th International Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC PART) / ICST 2018*, April 9th 2018.

[20] 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.

[21] 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.

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

[23] 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.

[24] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(1):1–36, 2022.