Model-based reasoning using answer set programming

Franz Wotawa1 · David Kaufmann1

Accepted: 18 January 2022 / Published online: 9 April 2022 © The Author(s) 2022

Abstract
Diagnosis, i.e., the detection and identification of faults, provides the basis for bringing systems back to normal operation in case of a fault. Diagnosis is a very important task of our daily live, assuring safe and reliable behavior of systems. The automation of diagnosis has been a successful research topic for several decades. However, there are limitations due to complexity issues and lack of expressiveness of the underlying reasoning mechanisms. More recently logic reasoning like answer set programming has gained a lot of attention and practical use. In this paper, we tackle the question whether answer set programming can be used for automating diagnosis, focusing on industrial applications. We discuss a formalization of the diagnosis problem based on answer set programming, introduce a general framework for modeling systems, and present experimental results of an answer set programming based diagnosis algorithm. Past limitations like not being able to deal with numerical operations for modeling can be solved to some extent. The experimental results indicate that answer set programming is efficient enough for being used in diagnosis applications, providing that the underlying system is of moderate size. For digital circuits having less than 500 components, diagnosis time has been less than one second even for computing triple fault diagnoses.

Keywords Model-based diagnosis · Answer set programming · Modeling for diagnosis · Experimental evaluation

1 Introduction

In reasoning, we rely on previously obtained knowledge, for solving certain tasks like prediction or explanation. The former deals with making educated guesses like whenever I am turning on the ignition switch in a car the motor will start, enabling me to drive from my home to any destination. The latter is used for giving a reasonable explanation of how systems work, or giving a reason for an observation contradicting our expectations. For example, we may want to know why the car’s motor is not running after turning on ignition. In this work, we focus on the explaining such situations making use of prior knowledge to be formalized in logic. Model-based reasoning makes use of a model of a system for explaining discrepancies between given observations and the expected behavior of the system. This type of reasoning is called reasoning from first principles (see e.g. [9, 48]).

Diagnosis and configuration, which are related, have been application areas of interest for artificial intelligence since its beginnings. Systems like MYCIN [3] or ARTEX [16] utilize logic-based reasoning for explaining misbehavior. However, in contrast to model-based reasoning those systems require diagnosis knowledge to be formulated in the direction from symptoms (or effects) to causes, making them less flexible and easy to adapt for diagnosing different systems. Model-based reasoning can be seen as an answer to the underlying challenge of making use
of system models, which are usually given in the direction from causes to effects or in a relational form, directly. As a consequence, models capturing the structure and behavior of systems can be more easily adapted in case of system changes or when being applied for modeling other similar systems and, therefore, reducing development and maintenance costs. Model-based diagnosis has been shown to be applicable in many domains including power transmission networks [2, 46], wind turbine diagnosis [34], automotive [38, 42, 44, 49], space [41, 45] and more recently [5], configuration [13, 52], programs [20, 37, 58], and even spreadsheets [1, 27, 29].

In order to apply model-based reasoning in practice, we require a fast and expressive theorem prover, providing enough expressiveness to formalize relevant knowledge, and to be fast enough for coming up with diagnostic explanations within a reasonable amount of time. Note that it is difficult to state a time limit for diagnosis because it is very much dependent on the application area. A diagnosis application used in a car driving at 60 miles per hour needs to be faster for preventing a crash, than an application of diagnosis for post-mortem design analysis, which may require hours or days. Moreover, the overall diagnosis time depends on the models used, the size of the models reflecting the size of corresponding systems, as well as the particular diagnosis problem. However, in any case for a reasoning engine to be used we would expect to return results within seconds even for mid-size systems having between 500 and 1,000 components.

In this paper, we focus on the question whether theorem provers relying on answer set programming [10, 15] can be effectively used for diagnosis based on models. In the particular case, we assume that the models are formulated in a language current answer set programming implementations are using. A diagnosis algorithm makes use of this model, and interacts with the theorem prover for computing diagnoses. We discuss such an algorithm in the paper, and show how it can be coupled with an answer set prover. We also introduce the coding of models in a general way that can be adapted for many different applications ranging from Boolean to analog circuits. Moreover, we present experimental results considering concrete circuits and discuss the obtained runtime results in detail. Using the experimental results, we are able to answer the question whether and in which context under given requirements answer set programming can be effectively used for model-based diagnosis.

This paper is a substantially modified and extended version of [56]. In contrast to our previous paper, we (1) switch focus on modeling for diagnosis utilizing answer set programming considering different application scenarios, (2) discuss more examples and details, and (3) present the results of a more detailed experimental evaluation. We structure the paper as follows: In Section 2 we introduce the basic foundations behind model-based reasoning and answer set programming. Afterwards, in Section 3, we introduce modeling considering Boolean and numeric circuits, discussing abstraction to be used for diagnosis, and show how these models can be represented using answer set programming in a unified way. In Section 4, we present experimental results obtained when using answer set programming for diagnosis with the purpose of showing practical applicability. After discussing related research in Section 5, we conclude the paper.

2 Basic foundations

To be self-contained we are going to discuss the underlying basic foundations including definitions and algorithms in this section. We first start discussing the principles of model-based diagnosis following Reiter [48]. Afterwards, we introduce the basic concepts behind answer set programming [10, 15] and how they can be used for computing diagnoses.

2.1 Model based reasoning

The underlying idea behind model-based reasoning [9, 33, 48] has always been to solve given problems, considering the available knowledge directly. In case of model-based diagnosis, the focus has been on the identification of causes behind the observed symptoms or effects in order to repair a system or heal a patient. In contrast to previous work on diagnosis, diagnoses should be derived from models directly without requiring to reformulate knowledge specifically to fit a certain theorem prover or other reasoning engine. The model-based diagnosis principle can best be explained, using an example. Let us have a look at the half adder circuit depicted in Fig. 1 comprising 5 digital NAND gates. A half adder is for adding Boolean numbers (in our case \( a \) and \( b \)). The result is stored in \( s \) and \( c \), where \( s \) stores the sum, and \( c \) represents the carry bit required if we add 1 and 1, which equals to 10 in binary format. The expected behavior of the half adder is depicted on the right of Fig. 1.

Let us assume that we want to add 0 (for \( a \)) and 1 (for \( b \)), but receive a 1 for \( c \) and a 0 for the output \( s \). Hence, both values of outputs are wrong, and we are interested in explaining the observations. In particular, we want to identify components of the circuit that behave faulty in order to replace them for bringing the half adder back to normal operation. Setting the health state of components to faulty would be a cause explaining the given observations. But how to obtain such diagnoses? What we know is the behavior of the components, i.e., the NAND gates in our
example, and the structure. Let us use this knowledge for identifying the diagnoses.

We start assuming that all components behave correctly, i.e., their health state is considered not being abnormal. In this case, we are able to compute a value of 1 at the output of component N1. Taking this value and using the behavior of gate N5 leads to the conclusion that the output of N5 must be 0 contradicting the observation hat c is 1. Hence, the assumption that all components work cannot be true. Let us therefore assume that all components except N1 are correct.

Using the observed output of N5, which is 1, we know that the input of N5 must be 0. Because of this, the output of both gates N2 and N3 must be 1, leading finally to the output value 0 for gate N4, which we observe. Hence, there is no contradiction anymore, and we are able to conclude that component N1 is a diagnosis explaining the given observations. When continuing this process, i.e., selecting certain components to be faults (or behaving abnormally) and assuming all other components to work as expected, we would be able to identify two other diagnoses N3, N5 and N4, N5 that require 2 components to fail at the same time.

In the following, we make use of the underlying idea of explicitly using the health state of a component, when formalizing diagnosis, but first, we start with discussing models, where we rely on first order logic (see e.g., [4]) considering predicates \( p(.) \) (with functors and variables), the logical operators NOT (\( \neg \)), AND (\( \land \)) and implication \( \rightarrow \), and the all quantifier (\( \forall \)). We say that a logical sentence is ground if it does not comprise any variables, and we assume that every variable in a logical formula has a corresponding quantifier. Moreover, we only consider facts \( p \), where \( p \) is a grounded predicate, and rules of the form \( \forall X_1 \ldots \forall X_m : p_1 \land \ldots \land p_n \rightarrow p_{n+1} \) where all \( p_i \)'s are literals, i.e., either a predicate or a negated predicate, and \( X_1 \) to \( X_m \) are variables used in the literals of the rule. Furthermore, we assume that \( p_{n+1} \) must be a predicate (and not comprise \( \neg \)) or \( \bot \) stating inconsistency.

The semantics of a rule \( \forall X_1 \ldots \forall X_m : p_1 \land \ldots \land p_n \rightarrow p_{n+1} \) is given using an interpretation, i.e., a function mapping grounded predicates to a truth value true (\( \top \)) or false (\( \bot \)). Note that grounding replaces variables with constants. Therefore, after grounding predicates can be considered as ordinary propositions. If all literals \( p_1 \) to \( p_n \) on the left side of a rule become true, then \( p_{n+1} \) must be true. For more details about the semantics, we refer the interested reader to any textbook on classical first order logic (e.g., [4]). However, for the context of this paper it is sufficient to assume the existence of a theorem prover, that checks whether a given logical sentence is consistent (or satisfiable) and to focus on the description of models.

It is also worth noting, that model-based diagnosis is not restricted to any type of logic including first-order logic for representing models and computing diagnoses. Any means for reasoning allowing to check consistency can be used.

For coming up with a (logical) model for our half adder circuit, we introduce required predicates. First of all, we follow the principle of coming up with a model of components and their connections. Components are connected via their ports, where we do not distinguish input form output ports. The ports provide the interface between the component model and its connections. The component behavior is defined considering values that are visible on the ports. Second, and because of the used component-connection model, we make use of the following predicates: (i) \( \text{val}(p(c), v) \) stating a value \( v \) on a port \( p \) of component \( c \), (ii) \( \text{conn}(p_1, p_2) \) defining a connection between two ports \( p_1 \) and \( p_2 \) of not necessarily different components, and (iii) \( \text{ab}(c) \) for indicating that component \( c \) is faulty, i.e., behaving abnormally. Furthermore, we do not want to define the behavior for each component separately. The half adder circuit from Fig. 1 comprises 5 NAND gates, where each behave in the same way (when assuming to work correctly). Hence, we make use of a predicate \( \text{type}(c, t) \) for a component \( c \) stating that it belongs to the type \( t \), e.g., NAND. We will make use of this predicate to define the
behavior for a certain type and for setting the type for a particular component.

Using these predicates, we are able to formalize the behavior of a NAND gate as follows:

\[
\begin{align*}
\forall C : & \text{ type}(C, \text{nand}) \land \neg \text{val}(C) \land \text{val}(\text{in}(C), 1) \land \text{val}(\text{in}(C), 0) \rightarrow \text{val}(\text{out}(C), 0) \\
\forall C : & \text{type}(C, \text{nand}) \land \neg \text{val}(C) \land \text{val}(\text{in}(C), 0) \rightarrow \text{val}(\text{out}(C), 1) \\
\forall C : & \text{type}(C, \text{nand}) \land \neg \text{val}(C) \land \text{val}(\text{in}(C), 2) \rightarrow \text{val}(\text{out}(C), 1) \\
\forall C : & \text{type}(C, \text{nand}) \land \neg \text{val}(C) \land \text{val}(\text{out}(C), 0) \rightarrow \text{val}(\text{in}(C), 1) \\
\forall C : & \text{type}(C, \text{nand}) \land \neg \text{val}(C) \land \text{val}(\text{out}(C), 1) \rightarrow \text{val}(\text{in}(C), 0) \\
\forall C : & \text{type}(C, \text{nand}) \land \neg \text{val}(C) \land \text{val}(\text{out}(C), 0) \rightarrow \text{val}(\text{in}(C), 1)
\end{align*}
\] (1)

The behavior captures the truth table entries for a NAND gate assuming that the gate is working correctly, i.e., \(\neg \text{val}(\cdot)\) must be true. When constructing the rules, we also have to take care of being unique. For example, if we know that the output is 1 and one input port is 0, we cannot determine the value of the other input port. After specifying the behavior of a component, we introduce two rules defining the propagation of values over connections:

\[
\begin{align*}
\forall P1 \forall P2 \forall V : & \text{conn}(P1, P2) \land \text{val}(P1, V) \rightarrow \text{val}(P2, V) \\
\forall P1 \forall P2 \forall V : & \text{conn}(P1, P2) \land \text{val}(P2, V) \rightarrow \text{val}(P1, V)
\end{align*}
\] (2)

\[
\begin{align*}
type(n1, \text{nand}) & \text{type}(n2, nand) \text{type}(n3, \text{nand}) \text{type}(n4, nand) \text{type}(n5, \text{nand}) \\
\text{conn}(a, \text{in}(n1)) & \text{conn}(b, \text{in}(n1)) \text{conn}(a, \text{in}(n2)) \text{conn}(\text{out}(n1), \text{in}(n2)) \\
\text{conn}(\text{out}(n1), \text{in}(n3)) & \text{conn}(b, \text{in}(n3)) \text{conn}(\text{out}(n2), \text{in}(n4)) \\
\text{conn}(\text{out}(n3), \text{in}(n4)) & \text{conn}(\text{out}(n1), \text{in}(n5)) \text{conn}(\text{out}(n1), \text{in}(n5)) \\
\text{conn}(\text{out}(n4), s) & \text{conn}(\text{out}(n5), s)
\end{align*}
\] (4)

In Eq. 4 the first row declares all components \(n1 \ldots n5\) to be of type NAND. The other 4 rows are for specifying the connections between the ports. There we use \(a, b\) and \(s, c\) as names for interfaces between the half adder and its outside world. For diagnosis, we make use of the model and its components, which are used to explain a detected deviation from expectations. Formally, we introduce the notion of a diagnosis system comprising the first part of necessary formalized knowledge for diagnosis as follows:

**Definition 1 (Diagnosis system)** A tuple \((SD, COMP)\) is a diagnosis system where \(SD\) is a model of the system (i.e., its system description), and \(COMP\) the set of components.

For the half adder circuit \(SD\) comprises all rules and facts stated in Eqs. 1 to 4. The set \(COMP\) is \(\{n1, n2, n3, n4, n5\}\).

Besides the model, we need observations for diagnosis. In case of the half adder, we formalize that \(a = 0, b = 1, s = 0\) and \(c = 1\) using the predicate \(\text{val}(\cdot)\) as follows:

\[
\text{val}(a, 0) \text{val}(b, 1) \text{val}(s, 0) \text{val}(c, 1)
\] (5)

The diagnosis system together with the observations now forms the diagnosis problem:

**Definition 2 (Diagnosis problem)** Let \((SD, COMP, OBS)\) be a diagnosis system and \(OBS\) a set of observations. The tuple \((SD, COMP, OBS)\) specifies a diagnosis problem that relies on finding explanations for the given observations.

A solution to the diagnosis problem is a diagnosis or a set of diagnoses, which are themselves set of components. Note that in the context of model-based diagnosis, we want to identify those components that are responsible for deviations between the observation and the expected behavior. Such components are considered to work as wrongly, i.e., abnormal. Hence, their faulty behavior explains the given observations.

**Definition 3 (Diagnosis)** Given a diagnosis problem \((SD, COMP, OBS)\). A set \(\Delta \subseteq COMP\) is a diagnosis, if and only if \(SD \cup OBS \cup \{\text{val}(C) | C \in \Delta\} \cup \{\neg \text{val}(C) | C \in COMP \setminus \Delta\}\) is consistent. A diagnosis \(\Delta\) is said to be
minimal (or parsimonious) if and only if there exists no set $\Delta' \subset \Delta$ that is itself a diagnosis.

Accordingly to this definition the sets $\{n1\}$, $\{n3, n5\}$, $\{n4, n5\}$ are minimal diagnoses of the half adder example. The set $\{n1, n2\}$ is also a diagnosis, because only assuming $n1$ to be faulty we remove all inconsistencies from the model together with the given observations. Hence, in practice it is sufficient to consider minimal diagnoses only. Note that this definition of minimality is different from others. In many cases, people would consider $\{n1\}$ to be the minimal diagnosis because of only comprising 1 component. Such a definition of minimality is called cardinality minimality considering the cardinality of the diagnoses. It is worth mentioning that in practice, we may also want to consider diagnoses accordingly to their cardinality, i.e., starting with diagnoses of size 1, and if not explaining the behavior correctly have a look at diagnoses of size 2, etc. until we find an explanation that is a valid one in the given context of diagnostic reasoning. How to distinguish diagnoses is a particularly important practical question and often rely on replacing components and having a look about consequences, or further measurements to be taken. In this paper, we are not further going to elaborate on this topic.

In the following, we outline an algorithm for computing diagnosis utilizing Definition 3 and the idea of computing minimal diagnoses with an increased size, which relies on [40]. The idea of $\text{IDIAG}$ is to start with a diagnosis of size 0, i.e., the empty diagnosis, where all components are assumed to behave correctly, and to increase diagnosis size until we do not compute any new minimal diagnoses. In order to avoid computing supersets of already computed diagnoses, we introduce further constraints into the model. These constraints formalize that superset are not wanted as solutions.

**Algorithm 1 $\text{IDIAG}(SD, COMP, OBS, n).$**

**Input:** An observation system $(SD, COMP)$, a set of observations $OBS$, and the desired cardinality $n$  
**Output:** All minimal diagnoses up to cardinality $n$

1. Let $Sols$ be $\{\}$
2. Let $M$ be $SD \cup OBS$.
3. for $i = 0 \rightarrow n$ do
4.    $S = \text{PROVER}(M, i, COMP)$
5.    if $i$ is 0 and $S$ is $\{\{\}\}$ then
6.        return $S$

7. end if  
8. Let $Sols$ be $Sols \cup S$.
9. for all $\{c_1, \ldots, c_i\}$ in $S$ do
10.     $M = M \cup \{ab(c_1) \land \ldots \land ab(c_i) \rightarrow \perp\}$.
11. end for
12. end for
13. return $Sols$

The $\text{IDIAG}$ algorithm takes a diagnosis problem $(SD, COMP, OBS)$ and the maximum cardinality $n$ of diagnosis and, first, generates a model $M$ in Line 2 comprising the system description $SD$ and the observations $OBS$. Afterwards it iterates over the diagnosis cardinality $i$ from 0 to the given maximum cardinality $n$. In each iteration, $\text{IDIAG}$ calls a theorem prover using the function $\text{PROVER}$ with the model $M$, the number of components that should be faulty $i$, and the set of components $COMP$ as arguments. This call is assumed to give back all diagnoses of size $i$ as result. Note that $\text{PROVER}$ maybe implemented generating all subsets of $COMP$ of size $i$ and checking consistency when assuming components of such a subset to be abnormal and the other to be working correctly. The function $\text{PROVER}$ may rely on a first order logic (FOL) theorem prover, a SAT solver, or even a general constraint solver. The only pre-requisite is that the underlying solver is able to use the model together with the observations for consistency checking. In the Section 3, we discuss an ASP realization of $\text{PROVER}$.

It is worth noting that in general $\text{PROVER}$ may not terminate depending on the underlying logic. For example, in FOL a theorem prover eventually halts with a contradiction. Hence, for the application of diagnosis, we assume that we use decidable fragments of FOL like propositional logic. Interestingly, ASP implementations make use of grounding for mapping the logical sentences of the theory comprising variables to their variable-free representation. Such a mapping – in the context of model-based diagnosis – usually not restricts generality because of the fact that the system structure as well as the domain can be determined.

After calling $\text{PROVER}$ the $\text{IDIAG}$ algorithm continues. In case of the first iteration where $i = 0$, we terminate if the empty set is a diagnosis (see Line 5), because in this case there are no further minimal diagnoses. Afterwards, in Line 8 the diagnoses of cardinality $i$ are added to the set of already computed solutions. In Line 10 new constraints are added stating that we are not interested in already obtained diagnoses and their supersets too.

Obviously, $\text{IDIAG}$ terminates with a result, which maybe empty when there is no minimal diagnosis of size $n$. The generated solutions only comprise minimal diagnoses up to cardinality $n$ due to the added constraints. The overall complexity of $\text{IDIAG}$ is of order $O(2^{COMP})$ when assuming that models only comprise fact and rules, because in the worst case we have to test all subsets of the set of $COMP$, which can be done in polynomial time under the given assumptions. However, in practice (and as we will see considering the use of ASP as theorem prover) diagnosis computation is much faster and allows even to diagnose systems of larger sizes.

There are many other algorithms for computing diagnosis. The original algorithm of Reiter [48] improved by
Greiner et al. [25] makes use of conflicts and hitting sets. A conflict is the dual concept of a diagnosis. Instead of searching for components that need to be faulty in order to retract inconsistencies, a conflict comprises all components that when assumed to be correct, lead to a conflict. Formally, a conflict is defined as follows:

**Definition 4 (Conflict)** Given a diagnosis problem \((SD, COMP, OBS)\). A set \(Co \subseteq COMP\) is a conflict, if and only if \(SD \cup OBS \cup \{\neg ab(C) | C \in Co\}\) is inconsistent. A conflict \(Co\) is said to be minimal (or parsimonious) if and only if there exists no set \(Co' \subset Co\) that is itself a conflict.

For our half adder example, the sets \(\{n1, n5\}\) and \(\{n1, n3, n4\}\) are both minimal conflicts. Conflicts can be often more easily computed using theorem provers. Hence, computing diagnoses from conflicts seems to be appealing. For doing this, we have to introduce the concept of hitting sets.

**Definition 5 (Hitting set)** Let \(F\) be a set of sets. A set \(H \subseteq \bigcup_{x \in F} x\) is a hitting set of \(F\) if for every \(x \in F\): \(x \cap H\) is not empty.

A hitting set “hits” all given sets. From [48] we know that all minimal diagnoses have to be minimal hitting sets of the set of all conflicts. The rational behind is that we have to set the health state of components in any conflict to be abnormal in order to resolve all contradictions. The original hitting set algorithm offers a lot of possibilities for improvement utilized in [53] and most recently in [43] to reduce diagnosis computation. For an empirical comparison of different diagnosis algorithm we refer to [39]. It is worth noting that there are other algorithms for diagnosis including [21] and more recently [12]. Furthermore, besides the discussed definition of diagnosis, which is also called consistency-based diagnosis, there is abductive diagnosis. For a more detailed description of similarities, differences, and corresponding work, we refer to [55, 57].

### 2.2 Answer set programming

After introducing the basics behind model-based diagnosis and some of the algorithms, we discuss answer set programming (ASP) [10, 15] and how it can be used for computing diagnoses using the \IDiAG algorithm. Note that the applicability of ASP for diagnosis can be derived from existing scientific literature. In [6] the authors showed how to implement default logic [47] using ASP. Because of the correspondence between default logic and diagnosis (see [48]), it is obvious that ASP can be used for implementing model-based diagnosis as well. However, in this subsection, we want to discuss how such an implementation can be done.

The concept of answer sets relies on the definition of stable models used to define a declarative semantics for logic programs implementing negation as failure. Negation as failure means that a certain negated predicate \(\neg p\) occurring in the right side of a rule (i.e., the body) can be interpreted as being correct, if the predicate cannot be derived from the other rules and facts. This type of negation is different because it does not require to derive the negated predicate or state it as fact. In the following we formally introduce stable models following the definition given in [36]. For the definitions we assume that we only have facts and rules as defined before, and that all predicates are grounded, i.e., do not comprise any variables. Furthermore, we assume a function \(Cons\) that takes a set of grounded predicates, and returns all predicates that can be derived from the predicates and the rules and facts of a given knowledge base.

**Definition 6 (Gelfond-Lifschitz transformation)** Given a set of rules and facts \(P\), and a set \(S\) of grounded predicates (or atoms, propositions). The Gelfond-Lifschitz transformation \(PS\) can be obtained as follows:

1. For each negative literal \(\neg q\) in the body of any rule in \(P\), if \(q \notin S\), then delete this literal from this body.
2. In the resulting set of rules, delete all those that still contain a negative literal in their bodies, i.e., if a rule contains a \(\neq q\) in its body such that \(q \in S\), then delete this rule.
3. Return the remaining transformed rules as result \(PS\).

Obviously \(PS\) does not comprise any negative literals anymore. When applying \(Cons\) to \(PS\) we obtain all predicates that can be derived from the transformed knowledge base. A stable model (or answer set) of the original knowledge base can be now defined as follows:

**Definition 7 (Stable model)** Given a set of rules and facts \(P\), and a set \(S\) of grounded predicates. \(S\) is a stable model (or answer set) if and only if \(S = Cons(PS)\).

Let us have a look at some examples. Let \(p, q \land p \rightarrow r, \neg q \land p \rightarrow s\) be a knowledge base. Do test whether \(\{p, s\}\) is an answer set, we first have to apply the Gelfond-Lifschitz transformation, resulting in \(p, q \land p \rightarrow r, p \rightarrow s\). From this set of rules, we can derive \(p\) and \(s\) but not \(q\) and \(r\). Hence, \(Cons(P_{\{p, s\}}) = \{p, s\}\) for the given knowledge base and \(\{p, s\}\) is an answer set. Obviously \(\{p, q, r\}\) is not an answer set because after the transformation we obtain the rules \(p, q \land p \rightarrow r\) from which we can only derive \(p\).
It is worth noting that a knowledge base may have no answer set. Let us consider the knowledge base: 
\[ \neg p \rightarrow p \]. \{ p \} cannot be an answer set, because after the transformation the only rule has to be deleted, and we cannot derive \( p \) from an empty knowledge base. Moreover, we may have more than one stable model. Let us consider the following knowledge base: 
\[ \neg q \rightarrow p, \neg p \rightarrow q. \]
\{\( p \)\} cannot be an answer set, because after the transformation the only rule has to be deleted, and we cannot derive \( p \) from an empty knowledge base. Moreover, we may have more than one stable model. Let us consider the following knowledge base:
\[ \neg q \rightarrow p, \neg p \rightarrow q. \] It is easy to see that \( \{ p \} \) and \( \{ q \} \) are both stable models. This last example, helps us formulating model-based diagnosis using answer sets. In particular, it allows us to come up with different models for one knowledge base, which is exactly what we require. For each component \( C \) we have a corresponding predicate \( ab(C) \), where its truth value has to be determined. For this purpose, we add another predicate \( nab(C) \) for a component \( C \) stating that \( C \) behaves not abnormal. We now can add two rules similar to the ones before:
\[ \neg ab(C) \rightarrow nab(C). \]
\[ \neg nab(C) \rightarrow ab(C). \] (6)

When using Eq. 6 together with the rest of the rules as described previously, we are able to find different answer sets all of them not being in conflict with the given observations. Hence, from each answer set, we are able to extract exactly one diagnosis as follows: Let \( AS \) be an answer set obtained using \( SD \) and \( OBS \), then \( \Delta = \{ C | \neg ab(C) \in AS \} \) must be a diagnosis of \( (SD, COMP, OBS) \). The set of all answer sets must be equivalent to the set of all minimal diagnoses as defined in this paper.

In the next section, we make use of a particular ASP implementation for coming up with diagnoses also explaining of how the ASP solver can be coupled with our IDIAG algorithm.

### 3 Modeling for diagnosis using ASP

ASP has gained a lot of attention over the past years because of its capabilities and the availability of fast tools. In particular, ASP tools allow to specify knowledge in a general form using predicates and variables and apply grounding, i.e., replacing variables with constants, before reasoning takes place. Hence, reasoning like checking for satisfiability can be done using efficient propositional SAT solvers. In the following, we explain how ASP can be used for computing diagnoses. We use the full adder example from Fig. 1 for illustrating the use of ASP where we rely on the input language of the ASP tool \( c\)lingo\(^1\) [22, 23], which is basically using an extended version of Prolog [7].

To convert facts and rules as introduced in the previous section, we only need to use the following transformation rules:

- A fact \( p \) is mapped to \( p. \)
- A rule \( p_1 \land \ldots \land p_n \rightarrow q \) is mapped to \( q : - p_1, \ldots, p_n. \)
- A rule \( p_1 \land \ldots \land p_n \rightarrow \bot \) is mapped to \( : - p_1, \ldots, p_n. \)

Variables used in rules have to be represented starting with a capitalized letter. The \( :- \) represents the implication \( \leftarrow \). At the end of a fact or rule there is always a \( \cdot \), and \( , \) represent the \( \land \) operator. When using this transformation rule, it is easy to represent the equations 1 to 3 as follows:

\begin{align*}
\text{val}(\text{out}(C, 0)) & : - \text{type}(C, \text{nand}), \text{nab}(C), \text{val}(\text{in1}(C), 1), \text{val}(\text{in2}(C), 1). \\
\text{val}(\text{out}(C, 1)) & : - \text{type}(C, \text{nand}), \text{nab}(C), \text{val}(\text{in2}(C), 0). \\
\text{val}(\text{out}(C, 1)) & : - \text{type}(C, \text{nand}), \text{nab}(C), \text{val}(\text{out}(C), 0). \\
\text{val}(\text{in1}(C, 0)) & : - \text{type}(C, \text{nand}), \text{nab}(C), \text{val}(\text{in1}(C), 1). \\
\text{val}(\text{in2}(C, 0)) & : - \text{type}(C, \text{nand}), \text{nab}(C), \text{val}(\text{out}(C), 1), \text{val}(\text{in2}(C), 1). \\
\text{val}(\text{out}(P, 0)) & : - \text{val}(\text{out}(P, 1)). \\
\text{val}(\text{out}(P, 0)) & : - \text{val}(\text{out}(P, 1)). \\
\end{align*}

In this representation, we replaced \( \neg ab(C) \) with \( nab(C) \) stating that component \( C \) is behaving not abnormally. This is due to the ASP representation of diagnosis where we consider \( ab(C) \) and its negated representation \( nab(C) \) for all components \( C \) as predicates that should be set by the ASP solver. For our model, we introduce the following rules for the predicates representing the health status of components (representing Eq. 6 using clingo syntax):

\begin{align*}
\text{comp}(X) & : - \text{type}(X, T). \\
\text{nab}(X) & : - \text{comp}(X), \text{not ab}(X). \\
\text{ab}(X) & : - \text{comp}(X), \text{not nab}(X). \\
\end{align*}

Note that this representation is very much general being able to handle different components. The first rule states that a component of any type is a component. If we come up with new types of components, we only need to add similar rules representing the behavior of the component (and not only the NAND gate). The last two rules state that either a component is ‘not abnormal’ (\( nab \)) in case it is not \( ab \), or it is ‘abnormal’ but only if it is not \( nab \).

What is missing is a possibility to state that we are only interested in faults of size $\$n\$. This is the case where there

---

\(^1\)https://potassco.org
is only one $ab$ predicate true. `clingo` provides means to handle this case. We are able to formulate a rule counting the number of $ab$’s. Via further constraining the number of $ab$’s we are able to restrict the ASP solver searching for single faults only. The following two rules server this purpose when using `clingo`:

```clingo
no_ab(N) :- N = #count { C : ab(C) }.
:- not no_ab($n$).
```

Note that before calling the ASP solver, we have to replace $n$ with a particular value. Moreover, the `clingo` representation described is generally valid for any system comprising components and connections. Later we will introduce additional rules representing analog and digital circuits.

In order to complete a model, we further have to add facts representing the structure of the circuit as well as observations. The `clingo` representation of Eq. 4 is:

```clingo
type(n1,nand).
type(n2,nand).
type(n3,nand).
type(n4,nand).
type(n5,nand).
conn(a,in1(n1)).
conn(b,in2(n1)).
conn(a,in1(n2)).
conn(b,in2(n2)).
conn(out(n1),in2(n2)).
conn(out(n1),in1(n3)).
conn(out(n2),in1(n3)).
conn(out(n3),in1(n4)).
conn(out(n4),in1(n5)).
conn(out(n1),in1(n5)).
conn(out(n5),c).
```

Observations from Eq. 5 are represented as follows:

```clingo
val(a,0).
val(b,1).
val(s,0).
val(c,1).
```

For any new diagnosis problem, we only need to adapt the structural description and the observations. The general definition of the behavior of components as well as rules required to compute diagnoses remain the same. Before discussing how to model different circuits, we describe how the function `PROVER` can be implemented using `clingo`.

**Algorithm 2 PROVER($M$, n, COMP).**

| Input: A model $M$, the size of diagnoses $n$, and a set of components `COMP`
| Output: All diagnoses of cardinality $n$ |

1: Let $M'$ be the `clingo` representation of model $M$ where we replaced $n$ with the value $n$.
2: Call `clingo` using $M'$ obtaining a set of answer sets $\Sigma$.
3: Let $Sols$ be the empty set.
4: for all $AS \in \Sigma$ do
5:    Add $\{C|ab(C) \in AS\}$ to $Sols$.
6: end for
7: return $Sols$

This algorithm takes the model and transfers it into a representation `clingo` can understand. The size of diagnoses to be obtained is set to the given value, and all answer sets are converted into a set of components that have to be faulty in order to remove inconsistencies. In practice, the `clingo` representation will be computed only once and successively used during diagnosis. We may also change Line 10 to represent already obtained diagnoses using the syntax of `clingo`.

### 3.1 Modeling circuit’s components

After explaining how to represent diagnosis models in ASP and in particular `clingo`, we now discuss further modeling issues. We still remain using the component connection model, requiring us only to define the behavior of the component, which logically represent the communication between interconnected components using ports. We restrict our view on digital and analog circuits where the latter is supposed to implement basic computations like the product or sum only. We further do not represent time in models. However, via adapting the representation of values using the `val` predicate, we would be able also to consider (discrete) time in our models.

We first start with presenting models for digital gates like inverters, buffers, AND, OR, NAND, NOR, XOR, and XNOR, having one or two inputs. The models can be extended to handle components with more inputs as well. In Fig. 2 we summarize the behavioral models. In contrast to the previously used model of a NAND gate, we now declare the behavior using tuple sets. This allows easily to specify at least the value propagation from the inputs to the outputs providing that there is a mathematical function define. Unfortunately, for the other direction, e.g., knowing
the output value and maybe one input (and except for the case of components with 1 input), we have to handle specific cases separately.

For example, let us consider the AND gate. If we know that the output is 1, then both inputs have to be 1 as well. However, if the output is 0, we cannot decide one input

| Model-based reasoning using answer set programming | Fig. 2 clingo models for digital circuits |
|---------------------------------------------------|--------------------------------------|
| tuple (buff, 0, 0).                             | val(out(C), Z) :- type(C, nand),      |
| tuple (buff, 1, 1).                             |   val(in1(C), V), val(in2(C), W),    |
| tuple (inv, 0, 1).                              |   tuple(nand, V, W, Z), nab(C).       |
| tuple (inv, 1, 0).                              | val(in1(C), 1) :- type(C, nand),     |
| tuple (and, 0, 0, 0).                           |   val(out(C), 0), nab(C).             |
| tuple (and, 1, 0, 0).                           | val(in2(C), 1) :- type(C, nand),     |
| tuple (and, 0, 1, 0).                           |   val(out(C), 0), nab(C).             |
| tuple (and, 1, 1, 1).                           | val(in1(C), 0) :- type(C, nand),     |
| tuple (nand, 0, 0, 1).                          |   val(out(C), 1), val(in2(C), 1),    |
| tuple (nand, 1, 0, 1).                          |     nab(C).                           |
| tuple (nand, 0, 1, 1).                          | val(in2(C), 0) :- type(C, nand),     |
| tuple (nand, 1, 1, 1).                          |   val(out(C), 1), val(in1(C), 1),    |
| tuple (or, 0, 0, 0).                            |     nab(C).                           |
| tuple (or, 1, 0, 1).                            | val(out(C), Z) :- type(C, or),       |
| tuple (or, 0, 1, 1).                            |   val(in1(C), V), val(in2(C), W),    |
| tuple (or, 1, 1, 1).                            |   tuple(or, V, W, Z), nab(C).         |
| tuple (nor, 0, 0, 1).                           | val(in1(C), 0) :- type(C, or),       |
| tuple (nor, 1, 0, 0).                           |   val(out(C), 0), nab(C).             |
| tuple (nor, 0, 1, 0).                           | val(in2(C), 0) :- type(C, or),       |
| tuple (nor, 1, 1, 0).                           |   val(out(C), 0), nab(C).             |
| tuple (xor, 0, 0, 0).                           | val(in1(C), 1) :- type(C, xor),      |
| tuple (xor, 1, 0, 1).                           |   val(out(C), 0), val(in2(C), 0),    |
| tuple (xor, 0, 1, 1).                           |     nab(C).                           |
| tuple (xor, 1, 1, 0).                           | val(out(C), Z) :- type(C, or),       |
| tuple (xnor, 0, 0, 1).                          |   val(in1(C), V), val(in2(C), W),    |
| tuple (xnor, 1, 0, 0).                          |   tuple(xnor, V, W, Z), nab(C).       |
| tuple (xnor, 0, 1, 0).                          | val(in1(C), 0) :- type(C, xor),      |
| tuple (xnor, 1, 1, 1).                          |   val(out(C), 1), nab(C).             |
| % Declaring component behavior                 | val(in1(C), 1) :- type(C, xor),      |
| % 1 input components                           |   val(out(C), 0), val(in2(C), 0),    |
| val(out(C), Z) :- type(C, T), val(in(C), Y),   |     nab(C).                           |
|   tuple(T, Y, Z), nab(C).                       | val(out(C), Z) :- type(C, T), val(   |
| val(in(C), Y) :- type(C, T), val(out(C), Z),   |   in(C), Y), tuple(T, Y, Z), nab(C).  |
|   tuple(T, Y, Z), nab(C).                       | val(out(C), Z) :- type(C, and),      |
| % 2 inputs components                          |   val(in1(C), V), val(in2(C), W),    |
| val(out(C), Z) :- type(C, and),                |   tuple(and, V, W, Z), nab(C).        |
|   val(in1(C), 1) :- type(C, and),              | val(in1(C), 1) :- type(C, and),      |
|   val(out(C), 1), nab(C).                       |   val(out(C), 1), nab(C).             |
| val(in2(C), 1) :- type(C, and),                | val(in1(C), 1) :- type(C, and),      |
|   val(out(C), 1), nab(C).                       |   val(out(C), 1), nab(C).             |
| val(in1(C), 0) :- type(C, and),                | val(in1(C), 0) :- type(C, and),      |
|   val(out(C), 0), val(in2(C), 1),              |   val(out(C), 0), val(in2(C), 1),    |
|     nab(C).                                     |     nab(C).                           |
| val(in2(C), 0) :- type(C, and),                | val(in1(C), 0) :- type(C, xnor),     |
|   val(out(C), 0), val(in1(C), 1),              |   val(out(C), Z), val(in2(C), W),    |
|     nab(C).                                     |   tuple(xnor, V, W, Z), nab(C).       |
| val(out(C), Z) :- type(C, xnor),               | val(in2(C), W) :- type(C, xor),      |
|   val(in1(C), V), val(in2(C), W),              |   val(out(C), Z), val(in1(C), V),    |
|   tuple(xnor, V, W, Z), nab(C).                 |   tuple(xnor, V, W, Z), nab(C).       |
| val(in1(C), V) :- type(C, xnor),               | val(in1(C), V) :- type(C, xnor),     |
|   val(out(C), Z), val(in2(C), W),              |   val(out(C), Z), val(in2(C), W),    |
|   tuple(xnor, V, W, Z), nab(C).                 |   tuple(xnor, V, W, Z), nab(C).       |
| val(in2(C), W) :- type(C, xor),                | val(in1(C), V) :- type(C, xnor),     |
|   val(out(C), Z), val(in2(C), W),              |   val(out(C), Z), val(in1(C), V),    |
|   tuple(xnor, V, W, Z), nab(C).                 |   tuple(xnor, V, W, Z), nab(C).       |
if we know that the other is 0. In this case the unknown input maybe 1 or 0. Hence, the rules handling this behavior have to be adapted to handle this specific cases. Note that whenever we cannot determine a value, we are not going to add a rule.

In a similar way, we can introduce models for analog circuits, where values can range from a certain value (e.g., 1 or -20) to another value (e.g., 10 or 100). In any case, we have to define again the tuples that represent the ordinary behavior of such gates implementing product or sum of numbers as functions. Of course, it would be also important to handle floating point numbers. However, we restrict our view to the restricted integer domain and show how the previous rules stating behavior have to be adapted. In Fig. 3 the clingo program implementing the behavior of the sum and product functions is given. Moreover, we have to change the constraint for values to be propagated from one port to another. In case of Boolean values, we are able to consider only the one case, i.e., that the value of a port can either be 0 or 1. However, in analog circuits, we have to state that there is only one value on a particular port. This is stated using the constraint $\text{val}(X,W), \text{val}(X,V), X\neq V$. stating that having two different values $W$ and $V$ on the same port $X$ leads to a contradicting.

The behavior of sum plus and the product mult functions are stated similarly to the Boolean circuit gates. We make use of tuples specifying the functions. In case of clingo we can rely on specific operators $+$ and $\ast$ for coming up with all of such tuples between a pre-specified minimum and maximum value. We do this using predicates $\text{pmin}$, $\text{pmax}$ and $\text{mmin}$, $\text{mmax}$ for the sum and product function respectively. Note that before using the model, we have to replace $\text{pm}$, $\text{mn}$, $\text{mm}$ with specific integer values. The behavior of the functions simple can rely on the generated tuples with one exception. For the multiplication, we have to consider the special case of 0, where we may not be able to determine all input values once the output is known to be 0. Hence, this special case is treated using additional rules.

What we will see in the experimental evaluation that there is an influence of the chosen domain for the sum and product gates and the overall diagnosis performance. This is due to the fact that for each value combination stated in tuple there is a grounded predicate generated before computing the answer sets. Hence, increasing the domain increases the number of predicates and rules. As a consequence, we have to carefully select the right domain required for diagnosis. An analysis of values to be able to be observed for a given circuit seems to be required before making use of the models. Alternatively, we may consider a domain abstraction, where instead of using all values we either only rely on certain values like being smaller, equivalent, or larger than 0, or consider deviation models. The latter consider the direction of deviations between the expected and observed behavior and try to identify the reasons using this information. For a more detailed analysis of abstraction and abstract models as well as their application to diagnosis, we refer to [26–28, 54]. Using the concepts of modeling used in this paper, it is easy to implement these abstract models using clingo's syntax.

**Fig. 3** clingo models for analog circuits considering adding and multiplying

| Predicate                                      |
|-----------------------------------------------|
| $\text{pmin}($pm$)$.                        |
| $\text{pmax}($pm$)$.                        |
| $\text{mmin}($mm$)$.                        |
| $\text{mmax}($mm$)$.                        |
| $\text{tuple}(\text{plus},X,Y,X\ast Y):=X=M..N,Y=M..N,\text{pmin}(M),\text{pmmax}(N)$. |
| $\text{tuple}(\text{mult},X,Y,X\ast Y):=X=M..N,Y=M..N,\text{mmin}(M),\text{mmmax}(N)$. |
| $\text{val}(\text{out}(C),Z):=\text{type}(C,T),\text{val}(\text{in1}(C),V),\text{val}(\text{in2}(C),W),\text{tuple}(T,V, W,Z),\text{cab}(C)$. |
| $\text{val}(\text{in1}(C),V):=\text{type}(C,T),T=\text{mult},\text{val}(\text{out}(C),Z),\text{val}(\text{in2}(C),W),\text{tuple}(T,V,W,Z),\text{cab}(C)$. |
| $\text{val}(\text{in2}(C),W):=\text{type}(C,T),T=\text{mult},\text{val}(\text{out}(C),Z),\text{val}(\text{in1}(C),V),\text{tuple}(T,V,W,Z),\text{cab}(C)$. |
| $\text{val}(\text{in1}(C),V):=\text{type}(C,T),T=\text{mult},\text{val}(\text{out}(C),Z),Z=0,\text{val}(\text{in2}(C),W),\text{tuple}(T,V,W,Z),\text{cab}(C)$. |
| $\text{val}(\text{in2}(C),W):=\text{type}(C,T),T=\text{mult},\text{val}(\text{out}(C),Z),Z=0,\text{val}(\text{in1}(C),V),\text{tuple}(T,V,W,Z),\text{cab}(C)$. |
| $\text{val}(\text{in1}(C),0):=\text{type}(C,T),T=\text{mult},\text{val}(\text{out}(C),0),\text{val}(\text{in2}(C),V),V=0,\text{cab}(C)$. |
| $\text{val}(\text{in2}(C),0):=\text{type}(C,T),T=\text{mult},\text{val}(\text{out}(C),0),\text{val}(\text{in1}(C),V),V=0,\text{cab}(C)$. |
| $:=\text{val}(X,W),\text{val}(X,V),V=W$. |
Let us summarize modeling for diagnosis using ASP. Modeling relies on the component connection model, where components have a certain type, which is indicated using the predicate `type`, and are connected via parts, which are terms of the form `in(C)`, `in1(C)`, `in2(C)`, `out(C)`, ... for components `C`. Connecting two ports is done using the predicate `conn`. A model makes use of predicates `comp`, `ab`, `nab` for stating components and their health state. In addition, for each component type we have to specify the behavior, i.e., rules specifying how input and output ports are related using values. A value on a port is represented using the predicate `val`. All the rules for specifying components and their behavior as well as how to propagate values represent the general model `MG` to be used for a certain system like a digital system comprising gates. It is worth noting that with a general model, we can diagnose any system relying on pre-defined component types.

A specific model for a system `MS` makes use of the predicates `type` and `conn` to declare the involved components and how they are connected. A specific model usually does not comprise logical variables. The model used for diagnosis is the union of the specific model and the general model, i.e., `M = MS ⊔ MG`. We have discussed how to model systems comprising components with a behavior that can be expressed using Boolean or integer values. We have not considered time and any other types like real values, arrays, lists, or other structures. Discrete time shall be considered changing the `val` predicate and the behavior description of components via adding an additional parameter for specifying time. Structures maybe introduced using logical functions and predicates describing behavior. Real or other numeric values may require using specific theorem provers. Alternatively, we may consider also domain abstraction instead for coming up with a diagnosis model.

### 4 Experimental evaluation

In order to answer the question whether today’s most recent ASP solvers can be used for diagnosis in an efficient way, we carried out several experiments. All of them are based on diagnosis models following the principles described in this paper. We implemented the proposed algorithm IDIAG in Python 3.6 and used clingo version 5.4.0 as ASP solver, where we used the standard parameters. We performed the experiments on an Apple iMac Pro (2017) computer setup with an Intel Xeon W 3 Ghz 10-core processor, 128 GB DDR4 2666 MHz memory and the up-to-date operating system macOS 10.14.6. We did not specify a time out limit but we limited search to obtain 100 diagnoses in the maximum. This limit was set due to the fact that in practice the interest in a set of diagnoses beyond 100 is not given.

For the experiments we carried out three types of experiments. The first used the famous ISCAS circuits 2. The circuits implement Boolean circuits ranging from 160 to 3,512 gates and which have been often used for comparing diagnosis algorithms (see e.g. [39]). For each circuit, 300 different test cases are available considering single, double, and triple faults as valid explanations. In Table 1 we summarize the statistics and implemented functions of the ISCAS circuits. It is worth noting that the results have been already published partially in [56] and in more detail in [30]. However, we add more information to the experimental results in this paper and discuss the results obtained from a different perspective. We use the first type of experiments to answer research question RQ1 Is IDIAG in combination with ASP faster than specialized diagnosis algorithms? For this purpose, we compare the runtime of IDIAG with a Java implementation of Reiter’s hitting set algorithm [25, 48] HS-DIAG, and with an

2See [http://www.cbl.ncsu.edu:16080/benchmarks/ISCAS85/](http://www.cbl.ncsu.edu:16080/benchmarks/ISCAS85/)
IDIAG implementation that relies on the MINION constraint solver [24], which we call IDIAG\textsubscript{MINION}. It is worth noting that both algorithms were already used in [39] where the authors showed that these two algorithms have a superior runtime when being used for computing single faults. Furthermore, for both algorithms there are internal time out limits specified.

After comparing different diagnosis algorithms, we focus on the capabilities and limitations of IDIAG based on ASP in the other experiments. For the second type of experiment, we wanted to gain further knowledge about the performance of ASP-based diagnosis depending on the number of components. Hence, we used inverter chains ranging from 10 to 10,000 inverters. For each inverter chain we only considered one test case.

In the third class of experiments, we focused on analog circuits to show that diagnosis using ASP can also be used effectively for systems requiring integer operations. The analog circuits are from a generic class of circuits, having several layers of components making use of one type of numeric operation that are coupled with the next layer reducing the number of outputs by 1 in each layer. For each of such circuits, we manually generated 3–4 test cases with a different number of failing outputs. In Fig. 4 we depict one circuit having 5 product gates connected to the input and 2 outputs, which we call circuit\textsubscript{5,2}. Note that we always start with multiplications at the first level.

The second and third type of experiments are used to answer the following research questions: 

\textbf{RQ2} What is the performance of diagnosis considering models using numeric operators? and 

\textbf{RQ3} What are the limitations when using ASP for computing diagnosis?.

In the following, we discuss the obtained results in detail and answer the mentioned research questions.

### 4.1 Results obtained

In Table 2 we summarize the average runtime results of IDIAG for the ISCAS circuits. In this table we also included the number of rules generated for each circuit and the runtime results of the hitting set algorithm implementation and IDIAG\textsubscript{MINION} an implementation of IDIAG that is based on the MINION constraint solver. For all three implementations, we report the average runtime in seconds for computing single, double, and triple faults. In case of a time out (only for HS-DIAG and IDIAG\textsubscript{MINION}) the table includes a ‘-’. Figures in bold indicate the smallest runtime for a certain circuit.

We see that in the case of single faults HS-DIAG and IDIAG\textsubscript{MINION} always performed better than IDIAG on average. In case of double and triple faults IDIAG was always possible to compute a results (which was not the case for HS-DIAG IDIAG\textsubscript{MINION}) and always performed better with one minor exception of circuit \textsubscript{c432}. The figures indicate that IDIAG utilizing ASP provides a superior

---

**Fig. 4** A poly circuit making use of numeric gates having 5 input gates and 2 outputs
Table 2. Average runtime (in seconds) obtained using clingo applied to the ISCAS85 circuits (data obtained from [30]).

| Circuit | #Rules | IDIAG | MINION | HS-DIAG | IDIAG | MINION | HS-DIAG |
|---------|--------|-------|--------|---------|-------|--------|---------|
| c432    | 3,313  | 0.88  | 0.11   | 0.11    | 0.005 | 0.006  | 0.004   |
| c990    | 4,404  | 0.07  | 0.16   | 0.16    | 0.16  | 0.03   | 0.03    |
| c880    | 4,100  | 0.04  | 0.04   | 0.04    | 0.04  | 0.04   | 0.04    |
| c1355   | 4,600  | 0.03  | 0.03   | 0.03    | 0.03  | 0.03   | 0.03    |
| c1908   | 5,200  | 0.02  | 0.02   | 0.02    | 0.02  | 0.02   | 0.02    |
| c2670   | 5,800  | 0.01  | 0.01   | 0.01    | 0.01  | 0.01   | 0.01    |
| c3540   | 6,400  | 0.01  | 0.01   | 0.01    | 0.01  | 0.01   | 0.01    |
| c5315   | 7,000  | 0.01  | 0.01   | 0.01    | 0.01  | 0.01   | 0.01    |
| c6288   | 7,600  | 0.01  | 0.01   | 0.01    | 0.01  | 0.01   | 0.01    |
| c7552   | 8,200  | 0.01  | 0.01   | 0.01    | 0.01  | 0.01   | 0.01    |

In comparison with two other diagnosis algorithm implementations: HS-DIAG and IDIAG/HS-DIAG.

In Fig 5 we depict the average runtime in seconds for computing single faults when using IDIAG as a function of the number of logic rules. We see that – even in the case of a substantial increase of runtime after 40,000 rules – the overall runtime behavior seems to be within an acceptable range. Making use of these observation we are able to answer RQ1 as follows: ASP-based diagnosis is able to outperform specialized algorithms like HS-DIAG for diagnosis and also the same algorithm but utilizing a different reasoning engine like MINION in case of double and triple faults. In case of single faults, specialized diagnosis algorithms as well as other provers may perform better. However, IDIAG provides a more predictable runtime having no substantial outliers compared to hitting set computation. Moreover, for many application areas like diagnosis of digital circuits diagnosis based on ASP offers a sufficient runtime performance even for single fault diagnosis.

In the case of inverter chains, the overall picture remains the same. In Fig. 6 we see the average runtime in seconds for single fault diagnosis as a function of the number of inverters ranging from 10 to 10,000, where we carried out each diagnosis 10 times. Note that we are using a double logarithmic scale in this figure. In the particular case of inverter chains where the only output is faulty, all inverters can be responsible. Hence, we only have single fault diagnoses, and the number of diagnoses is always equivalent to the number of inverters. From the figure we see that performance when dealing with diagnoses of higher size. However, for single faults other diagnosis algorithms like HS-DIAG and underlying reasoning engines like MINION are faster.

In Fig 5 we depict the average runtime in seconds for computing single faults when using IDIAG as a function of the number of logic rules. We see that – even in the case of a substantial increase of runtime after 40,000 rules – the overall runtime behavior seems to be within an acceptable range. Making use of these observation we are able to answer RQ1 as follows: ASP-based diagnosis is able to outperform specialized algorithms like HS-DIAG for diagnosis and also the same algorithm but utilizing a different reasoning engine like MINION in case of double and triple faults. In case of single faults, specialized diagnosis algorithms as well as other provers may perform better. However, IDIAG provides a more predictable runtime having no substantial outliers compared to hitting set computation. Moreover, for many application areas like diagnosis of digital circuits diagnosis based on ASP offers a sufficient runtime performance even for single fault diagnosis.

In the case of inverter chains, the overall picture remains the same. In Fig. 6 we see the average runtime in seconds for single fault diagnosis as a function of the number of inverters ranging from 10 to 10,000, where we carried out each diagnosis 10 times. Note that we are using a double logarithmic scale in this figure. In the particular case of inverter chains where the only output is faulty, all inverters can be responsible. Hence, we only have single fault diagnoses, and the number of diagnoses is always equivalent to the number of inverters. From the figure we see that
the average runtime is more or less following a straight line, which indicates that there is a polynomial function returning the runtime given the number of inverters. This is also reasonable, because the number of (grounded) rules used for diagnosis also depends polynomial on the number of inverters.

In the third and last type of experiments we carried out diagnosis computations using IDIAG on the poly circuits. In Table 3 the number of input components (#I), the number of outputs (#O), the number of components in total (#C), and the number of rules generated by clingo (#R) are depicted. In Table 4 the obtained runtime results are given. For all circuits we have carried out diagnosis 10 times computing single, double and triple faults. For the runtime we report the minimum, maximum, average, median, and standard deviation of the 10 runs.

What we see from the results is that even in case of a smaller number of components, the diagnosis time can be quite large. This is due to the fact of rules required to formalize the numerical operations in logic, which depends on the domain to be considered. With the exception of the larger circuits where we have to come up with a domain size of up to 200, the diagnosis time is within an acceptable range of less than one second. For the largest considered circuits having 2, 3, or 4 outputs, diagnosis time varies between 6 and 580 seconds.

To clarify the dependency between the used domain size and diagnosis time, we carried out a further experiment. We took the circuit circuit_4_2 and varied the domain size between 100 and 700. The diagnosis time obtained is summarized in Table 5 and the average runtime is depicted in Fig. 7 where the y-axis is of logarithmic scale. The runtime obviously increases with the domain size. The increase seems to be somehow polynomial, which is acceptable for practical applications.

Let us now use the findings of the other experiments to discuss the remaining research questions. Regarding RQ2 we can state that the runtime performance for analog circuits depends on the domain size and the size of the circuits. The domain size should be as small as possible, in order to allow a fast computation. If this cannot be achieved diagnosis might be unacceptable slow and other types of models like qualitative models should be used. The dependency of between domain size and runtime for a given circuit seems to be polynomial. For answering RQ3 we have to summarize the limitations of the presented diagnosis approach. In case of digital circuits diagnosis time is comparable with the one obtained using specialized diagnosis algorithms. However, there are cases where ASP-based diagnosis is substantially requiring more time. Hence, for larger circuits or in cases where fast response is required specialized diagnosis algorithms or different reasoning engines may be used. In case of analog circuits there is a huge impact of the underlying domain size on the required diagnosis time. Hence, even for smaller circuits, diagnosis time may be far too high to be of use in industrial applications. It is required to keep the domain size as small as possible and ideally adapt this for particular situations. How to adapt domain size specifically for a particular application, is an open research question.

Threads to validity include as always, the limited number of considered systems and models. For allowing to generalize some of the mentioned findings, there is a need of carrying out more and different experiments. However, the provided analysis is far more sophisticated than previous ones. The obtained diagnosis runtime seems also to be reasonable considering the different factors. Moreover, there might be an influence on the results because of the underlying implementation of IDIAG, HS-DAG, IDIAG_MINION and the used computer equipment. The used implementation of HS-DAG and IDIAG_MINION have been

![Fig. 6](image-url) Average runtime for diagnosis (in seconds) of the inverter chain circuits ranging from 10 to 10,000 components

| Circuit  | #I | #O | #C | #R  |
|----------|----|----|----|-----|
| circuit_3_2 | 3  | 2  | 5  | 59,933 |
| circuit_4_2 | 4  | 2  | 9  | 20,199 |
| circuit_5_2 | 5  | 2  | 14 | 40,763 |
| circuit_4_3 | 4  | 3  | 7  | 85,324 |
| circuit_5_3 | 5  | 3  | 12 | 20,268 |
| circuit_6_3 | 6  | 3  | 18 | 4,560,830 |
| circuit_5_4 | 5  | 4  | 9  | 110,748 |
| circuit_6_4 | 6  | 4  | 15 | 45,337 |
| circuit_7_4 | 7  | 4  | 22 | 5,622,780 |
Table 4 Minimum, maximum, average, median, and standard deviation of the runtime (in seconds) obtained using clingo applied to the analog poly circuits

| Circuit | MIN   | MAX   | AVG   | MEDIAN | STDV |
|---------|-------|-------|-------|--------|------|
| single faults                      |       |       |       |        |      |
| circuit_3_2 | 0.077 | 0.102 | 0.086 | 0.080  | 0.009|
| circuit_4_2 | 0.179 | 0.224 | 0.205 | 0.213  | 0.016|
| circuit_5_2 | 0.623 | 9.262 | 6.290 | 9.049  | 4.065|
| circuit_4_3 | 0.105 | 0.122 | 0.112 | 0.112  | 0.004|
| circuit_5_3 | 0.120 | 290.070 | 72.008 | 0.188  | 126.022|
| circuit_6_3 | 11.372 | 66.319 | 24.266 | 11.673 | 22.191|
| circuit_5_4 | 0.140 | 0.162 | 0.148 | 0.147  | 0.006|
| circuit_6_4 | 0.228 | 0.453 | 0.333 | 0.333  | 0.077|
| circuit_7_4 | 15.233 | 809.286 | 580.160 | 752.631 | 332.090|
| double faults                       |       |       |       |        |      |
| circuit_3_2 | 0.118 | 0.146 | 0.129 | 0.123  | 0.010|
| circuit_4_2 | 0.464 | 0.524 | 0.488 | 0.486  | 0.017|
| circuit_5_2 | 1.628 | 15.230 | 10.378 | 14.481 | 6.264|
| circuit_4_3 | 0.160 | 0.186 | 0.170 | 0.170  | 0.006|
| circuit_5_3 | 0.409 | 440.862 | 109.280 | 0.546  | 190.814|
| circuit_6_3 | 18.498 | 84.522 | 34.451 | 19.258 | 27.002|
| circuit_5_4 | 0.213 | 0.248 | 0.224 | 0.224  | 0.009|
| circuit_6_4 | 0.873 | 1.423 | 1.178 | 1.217  | 0.196|
| circuit_7_4 | 24.346 | 1,954.494 | 1,229.977 | 1,475.520 | 786.932|
| triple faults                       |       |       |       |        |      |
| circuit_3_2 | 0.156 | 0.186 | 0.168 | 0.163  | 0.010|
| circuit_4_2 | 0.510 | 0.572 | 0.535 | 0.533  | 0.017|
| circuit_5_2 | 1.732 | 19.603 | 13.335 | 18.877 | 8.316|
| circuit_4_3 | 0.216 | 0.258 | 0.230 | 0.229  | 0.008|
| circuit_5_3 | 0.788 | 587.962 | 146.244 | 1.055  | 254.829|
| circuit_6_3 | 25.674 | 108.518 | 46.016 | 27.393 | 33.669|
| circuit_5_4 | 0.284 | 0.333 | 0.302 | 0.300  | 0.012|
| circuit_6_4 | 2.210 | 3.487 | 2.601 | 2.370  | 0.465|
| circuit_7_4 | 35.273 | 2,478.081 | 1,565.275 | 1,879.009 | 980.941|

for computing single, double, and triple faults respectively

Table 5 Minimum, maximum, average, median, and standard deviation of the runtime (in seconds)

| Domain | #Rules   | MIN   | MAX   | AVG   | MEDIAN | STDV |
|--------|----------|-------|-------|-------|--------|------|
| 100    | 20,199   | 0.213 | 0.222 | 0.217 | 0.218  | 0.003|
| 200    | 80,199   | 0.888 | 0.940 | 0.916 | 0.916  | 0.018|
| 300    | 180,199  | 2.046 | 2.174 | 2.097 | 2.103  | 0.041|
| 400    | 320,199  | 3.930 | 4.224 | 4.024 | 4.010  | 0.080|
| 500    | 500,199  | 5.907 | 6.301 | 6.052 | 6.046  | 0.107|
| 600    | 720,199  | 9.446 | 9.753 | 9.566 | 9.572  | 0.096|
| 700    | 980,199  | 12.297 | 12.773 | 12.532 | 12.567 | 0.158|

the domain size for circuit circuit_4_2 with one faulty output and computing single fault diagnosis only
previously used for comparing diagnosis algorithms. The computer equipment is not specific for the given purpose. Moreover, we will make the implementations as well as the models and the obtained raw data available for other researchers re-evaluating the experimental analysis.

5 Related research

The content of this paper is mostly related to [56] and more recently [30]. In [56] the author already discussed the use of ASP for diagnosis presenting the underlying concepts and also initial experimental results limited to single faults only. This paper is – as already mentioned in the introduction – an extension providing a far deeper discussing on the underlying foundations and modeling, and as well as a more advanced experimental analysis. In [30], the authors focus more on the use of model-based reasoning in the area of agents and cyber-physical systems. In this paper, we focus more on the discussion of modeling. In addition, the experimental analysis discussed in this paper extends the previous paper substantially. In particular, we discuss more results both in the domain of Boolean but also analog circuits. Furthermore, we present experimental results showing the dependency of diagnosis runtime on the size of the underlying value domain. Given these results, we conclude that diagnosis based on ASP seems to be fast enough for practical applications but also identify limitations, i.e., when dealing with variables having large domain sizes. This allows us to further decide on how to model systems in practice.

Regarding the use of logic theorem provers for diagnosis there has been a lot of previous work but not considering ASP. In [19] the authors used a Prolog implementation of model-based diagnosis. The seminal paper of Reiter [48] showed how diagnosis can be achieved using logical models. [33] provide a diagnosis approach based on the ATMS [31], which is a reasoning system focusing on assumptions to hold in order to maintain consistencies of logic formulas. [21] provided an iterative algorithm for improving diagnosis computations, where systems are assumed to have logic models. Later on, [11] provided a specialized algorithm for tree-structured systems, which has been improved by [51]. [50] provided a general framework combining [11] and [51] algorithms. Other algorithms often relying on general constraint solvers as a replacement of theorem provers include [12, 14, 40] showing that finding efficient diagnosis algorithms is an open research area. Depending on available reasoning mechanisms as well as other optimization criteria different solutions has been provided. In contrast to these papers, we do not focus on providing a diagnosis algorithm but trying to clarify the question of whether current ASP solvers can be used for diagnosis directly. The reported runtime results in this paper indicate that ASP solvers are fast enough for diagnosis and also provide enough expressiveness of the input language for modeling.

Alternative diagnosis algorithms like the one described in [13] deal with any-time algorithms, i.e., diagnosis algorithms that provide solutions until a time budget is consumed. This is different from our work, where we are interested in gaining a specific number of diagnosis and not only the first few ones. However, using ASP we can always set the limit of diagnoses to be computed and, therefore, also be able to stop diagnosis computation at a certain point in time. But of course, ASP solvers does not provide the same functionality than any-time algorithms allowing to come up with a (maybe not perfect) solution within a pre-defined time budget. In [35] the authors present an algorithm for diagnosis that is based on abductive reasoning requiring knowledge about the faulty behavior of components and systems. In this paper, however, we focus on consistency-based diagnosis, where we only require the correct behavior of components to be specified.

Modeling issues have been considered since the beginnings of model-based diagnosis research. [32] discussed the use and integration of fault models into consistency-based diagnosis potentially restricting the number of diagnoses that can be obtained from models. [18] showed that fault models are not necessarily required for restricting diagnoses to feasibly ones only. There the authors showed that knowledge about physical impossibilities can also allow to remove diagnosis candidates. [17] provided a general framework for abductive diagnosis that is an alternative diagnosis
characterization based on fault models only. [8] showed that even knowledge can be integrated into abductive diagnosis. Most recently [55, 57] discussed all different types of diagnosis considering recent application areas like autonomous systems. Integrate fault models into ASP-based diagnosis has not been discussed in this paper. For a discussion on this topic, we refer to [30]. However, we discussed different types of modeling, the obtained experimental evaluation results, as well as limitations, hence, complementing other papers dealing with the use of ASP for diagnosis.

In summary, this paper deviates from previous related research as follows: It provides theoretical foundations as well as a detailed experimental analysis showing applicability for a wider range of applications but also limitations, which have to be considered in practice. The underlying ASP theorem prover has a rich input language allowing to formalize a wide range of system models in a more intuitive way. Furthermore, the use of ASP avoids coming up with specific mappings from system models to their constraint or logic representation, and relying on specialized reasoning engines.

6 Conclusions

In this paper, we have discussed the use of answer set programming for model-based diagnosis. In particular, we introduced a way of formulating diagnosis relevant knowledge making use of answer set specific logical constructs. The underlying idea is to represent the health state of components as predicates that can be either abnormal or not abnormal. We formulated that the abnormal case can only be established when the normal case is not happening and vice versa. The corresponding two rules allow an answer set solver to select the health state accordingly to given observations such that we eliminate any inconsistency. Besides the formalization of the diagnosis problem as an answer set knowledge base we discuss modeling in more detail providing ways of dealing with digital and analog circuits. Furthermore, we present results obtained from an experimental evaluation. We refer to previous evaluations and extend them providing experiments for analog circuits and a way deeper analysis of the outcome. We also identified limitations and at least partially discussed some possible solutions like abstraction.

The obtained results indicate that answer set programming provide a good basis for the practical application of model-based diagnosis. Especially for digital circuits the experimental results are very much promising. For diagnosis dealing with double or triple faults the presented solution is superior. The provided experimental results for digital circuits confirm results published previously, i.e., [30]. However, we further added a comparison with another diagnosis algorithm, which is based on constraint solving. The obtained empirical results indicate that answer set programming is not only faster when searching for double or triple fault but is also able to solve more instances than the other diagnosis algorithms. In addition, we provided results for diagnosis of analog circuits, where we identified a critical relation between the used domain size and the diagnosis time required. The reason behind relies on the way answer set programming is handling logic variables, which are grounded, i.e., replaced with constants, before starting theorem proving. This leads to a substantially higher number of rules and as a consequence to a substantial increase of diagnosis computation. Finally, we showed that we are able to come up with analog circuit models very much similar to models used for digital circuits. Hence, a wide range of models can be implemented using the current capabilities of answer set programming tools.

Future research include improving modeling capturing qualitative models, the optimization of answer set solving, and finding the optimal domain size for analog circuits to make the models of such circuits more generally applicable in practice. Optimization answer set solving would include searching for optimal solving parameters in order to minimize diagnosis time. The parameters may depend on the structure of the model or any other model or system metric.

Funding Open access funding provided by Graz University of Technology.

Declarations

Conflict of interests The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abreu R, Hofer B, Perez A, Wotawa F (2014) Using constraints to diagnose faulty spreadsheets, Software Quality Journal pp 1–26. https://doi.org/10.1007/s11219-014-9236-4
2. Beschta A, Dressler O, Freitag H, Montag M, Struss P (1992) A model-based approach to fault localization in power transmission networks Intelligent Systems Engineering
3. Buchanan BG, Shortliffe EH (eds) (1984) Rule-based expert systems - The MYCIN experiments of the stanford heuristic programming project. Addison-Wesley Publishing Company, Boston

4. Chang CL, Lee RCT (1973) Symbolic logic and mechanical theorem proving academic press

5. Chanterry E, Trav-Massuyés L, Pencolé Y, Ferlac RD, Dellandréa B (2019) Applying active diagnosis to space systems by on-board control procedures. IEEE Trans. Aerosp. Electron. Syst. 55(5):2568–2580

6. Chen Y, Han H, Zhang Y, Zhou Y (2010) dl2asp: Implementing default logic via answer set programming. In: T. Janhunen, I. Niemel? (eds.) Logics in Artificial Intelligence, pp 104–116. Springer Berlin Heidelberg, Berlin, Heidelberg

7. Clocksin WF, Mellish CS (2003) Programming in Prolog, 5th edn Springer-Verlag

8. Console L, Torasso P (1990) Integrating models of correct behavior into abductive diagnosis. In: Proceedings of the European Conference on Artificial Intelligence (ECAI). Pitman Publishing, Stockholm, pp 160–166

9. Davis R (1984) Diagnostic reasoning based on structure and behavior. Artif Intell 24:347–410

10. Eiter T, Ianni G, Krenwallner T (2009) Answer set programming:

11. Fattah YE, Dechter R (1995) Diagnosing tree-decomposable circuits. In: Proceedings 14th International Joint Conf. on Artificial Intelligence, pp 1742–1748

12. Felfernig A, Schubert M, Zehentner C (2012) An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM 26(1):53–62. https://doi.org/10.1017/S0890060011000011

13. Felfernig A, Walter R, Galindo JA, Benavides D, Erdeniz SP, Atas M, Reiterer S (2018) Anytime diagnosis for reconfiguration. J Intell Inf Syst 51(1):161–182

14. Felfernig A, Walter R, Reiterer S (2015) Flexdiag: anytime diagnosis for reconfiguration. In: Configuration workshop, CEUR workshop proceedings, vol 1453, pp 105–110. CEUR-WS.org

15. Ferraris P, Lifschitz V (2005) Mathematical foundations of answer set programming. In: We will show them! (1), pp 615–664. College publications

16. Fleischanderl G, Friedrich G, Retti J (1989) ARTEX - Configuration-driven Diagnosis for routing systems. In: Proceedings of the Austrian Conference on Artificial Intelligence. Springer-Verlag

17. Friedrich G, Gottlob G, Nejdl W (1990) Hypothesis classification, abductive diagnosis and therapy. In: Proceedings of the International Workshop on Expert Systems in Engineering. Springer Verlag, Lecture Notes in Artificial Intelligence, Vo. 462, Vienna

18. Friedrich G, Gottlob G, Nejdl W (1992) Physical impossibility instead of fault models. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp 331–336. Boston (1990). Also appears in Readings in Model-Based Diagnosis (Morgan Kaufmann

19. Friedrich G, Nejdl W (1992) MOMO –Model-based Diagnosis for everybody. In: Proceedings of the IEEE Conference on Artificial Intelligence Applications (CAIA). Santa Barbara (1990). A slightly revised and extended version appears in Readings in Model-Based Diagnosis (Morgan Kaufmann

20. Friedrich G, Stumppner M, Wotawa F (1999) Model-based diagnosis of hardware designs. Artif Intell 111(2):3–39

21. Fröhlich P, Nejdl W (1997) A static model-based engine for model-based reasoning. In: Proceedings 15th International joint conf. on artificial intelligence. Nagoya, Japan

22. Gebser M, Kaminski R, Kaufmann B, Schaub T (2014) Clingo = ASP + control: Preliminary report. CoRR abs/1405.3694v10.3694

23. Gebser M, Kaminski R, König A, Schaub T (2011) Advances in gringo series 3. In: LPNMR, Lecture Notes in Computer Science, vol 6645, pp 345–351. Springer

24. Gent IP, Jefferson C, Miguel I (2006) Minion: a fast scalable constraint solver. In: ECAI, Frontiers in Artificial Intelligence and Applications, vol 141, pp 98–102. IOS Press

25. Greiner R, Smith BA, Wilkerson RW (1989) A correction to the algorithm in Reiter’s theory of diagnosis. Artif Intell 41(1):79–88

26. Hofer B, Hofer A, Wotawa F (2017) Combining models for improved fault localization in spreadsheets. IEEE Trans Reliab 66(1):38–53

27. Hofer B, Nica I, Wotawa F (2017) AI For localizing faults in spreadsheets. In: ICTSS, Lecture Notes in Computer Science, vol 10533, pp 71–87. Springer

28. Hofer B, Nica I, Wotawa F (2017) Qualitative deviation models for spreadsheet debugging. In: ISSRE Workshops, pp 191–198. IEEE computer society

29. Jannach D, Schmitz T (2014) Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. Automated Software Engineering pp 1–40. https://doi.org/10.1007/s10111-014-0141-7

30. Kaufmann D, Nica I, Wotawa F (2021) Intelligent agents diagnostics – enhancing cyber-physical systems with self-diagnostic capabilities. Advanced Intelligent Systems, 2000218. https://doi.org/10.1002/aisy.202000218. https://onlinelibrary.wiley.com/doi/abs/10.1002/aisy.202000218

31. de Kleer J (1986) An assumption-based TMS. Artif Intell 28:127–162

32. de Kleer J, Mackworth AK, Reiter R (1992) Characterizing diagnosis and systems. Artificial Intelligence 56

33. de Kleer J, Williams BC (1987) Diagnosing multiple faults. Artif Intell 32(1):97–130

34. Koitz-Hristov R (2018) From theory to practice: Abductive model-based diagnosis and its industrial application. Graz University of Technology, PhD Thesis

35. Koitz-Hristov R, Wotawa F (2020) Faster horn diagnosis - a performance comparison of abductive reasoning algorithms. Appl Intell 50(5):1558–1572

36. Lin F, Zhao Y (2004) Assat: Computing answer sets of a logic program by sat solvers. Artif Intell 157

37. Mayer W, Wotawa F (2021) Artificial intelligence methods for software debugging. In: M. Kalech, R. Abreu, M. Last (eds.) Artificial intelligence methods for software engineering. World Scientific. https://doi.org/10.1142/12360

38. Milde H, Guckenbiel T, Malik A, Neumann B, Strauss P (2000) Integrating Model-based diagnosis techniques into current work processes – Three case studies from the INDIA project. AI communications 13. Special Issue on industrial applications of model-based reasoning

39. Nica I, Pill I, Quaritsch T, Wotawa F (2013) The route to success instead of fault models. In: Proceedings of the National Conference on Artificial Intelligence Applications (CAIA). Santa Barbara (1990). A slightly revised and extended version appears in Readings in Model-Based Diagnosis (Morgan Kaufmann

40. Nica I, Wotawa F (2012) Condiag – computing minimal diagnoses for reconfiguration. In: Configuration workshop, CEUR workshop proceedings, vol 6645, pp 345–351. Springer

41. Pell B, Bernard D, Chien S, Gat E, Muscettola N, Nayak P, Wagner M, Williams B (1996) A remote-agent prototype
Model-based reasoning using answer set programming

for spacecraft autonomy. In: Proc. of the SPIE conference on optical science, engineering, and instrumentation, volume on space sciencecraft control and tracking in the new millennium. Bellingham, Washington, U.S.A., Society of Professional Image Engineers

42. Picardi C, Bray R, Cascio F, Console L, Dague P, Dressler O, Millet D, Rehfuß B, Struss P, Vallée C. (2002) Iddl: Integrating diagnosis in the design of automotive systems. In: Proceedings of the European Conference on Artificial Intelligence (ECAI). IOS Press, Lyon, France, pp 628–632

43. Piil I, Quaritsch T (2015) Rc-tree: a variant avoiding all the redundancy in reiter’s minimal hitting set algorithm. In: ISSRE Workshops, pp 78–84. IEEE computer society

44. Pons R, Subias A, Travé-Massuyés L. (2015) Iterative hybrid causal model based diagnosis:. Application to automotive embedded functions. Eng Appl Artif Intell 37:319–335

45. Rajan K, Bernard D, Dorais G, Gamble E, Kanefsky B, Kurien J, Millar W, Muscettola N, Nayak P, Rouquet N, Smith B, Taylor W, Tung Y (2000) Remote agent: an autonomous control system for the new millennium. In: Proceedings of the 14th European Conference on Artificial Intelligence (ECAI). Berlin, Germany

46. Rayudu RK (2010) A knowledge-based architecture for distributed fault analysis in power networks. Eng Appl Artif Intell 23(4):514–525. https://doi.org/10.1016/j.engappai.2010.02.007. https://www.sciencedirect.com/science/article/pii/S0952197610000655

47. Reiter R (1980) A logic for default reasoning. Artificial Intelligence 13(1–2)

48. Reiter R (1987) A theory of diagnosis from first principles. Artif Intell 32(1):57–95

49. Sachenbacher M, Struss P, Carlén C. M (2000) A Prototype for Model-based On-board Diagnosis of Automotive Systems. AI Communications 13. Special Issue on Industrial Applications of Model-Based Reasoning

50. Sachenbacher M, Williams BC (2004) Diagnosis as semiring-based constraint optimization. In: Proceedings of the 16th European Conference on Artificial Intelligence (ECAI). Valencia, Spain, pp 873–877

51. Stumptner M, Wotawa F (2001) Diagnosing tree-structured systems. Artif Intell 127(1):1–29

52. Tazl OA, Perko A, Wotawa F (2019) Conversational recommendations utilizing model-based reasoning. In: ConfWS, CEUR Workshop Proceedings, vol 2467, pp 13–19. CEUR-WS.org

53. Wotawa F (2001) A variant of reiter’s hitting-set algorithm. Inf. Process. Lett. 79(1):45–51

54. Wotawa F (2016) On the use of qualitative deviation models for diagnosis. In: Proceedings of the 29th International Workshop on Qualitative Reasoning. See https://ivi.fwi.uva.nl/ics/QRgroup/qr16/pdf/QR2016Proceedings.pdf

55. Wotawa F (2019) Sayed-Mouchaweh Reasoning from first principles for self-adaptive and autonomous systems. In: M. Sayed-Mouchaweh (eds.) Predictive Maintenance in Dynamic Systems – Advanced Methods, Decision Support Tools and Real-World Applications. Springer. https://doi.org/10.1007/978-3-030-05645-2

56. Wotawa F (2020) On the use of answer set programming for model-based diagnosis. In: H. Fujita, P. Fournier-Viger, M. Ali, J. Sasaki (eds.) 33rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE), LNCS vol 12144. Springer Nature Switzerland AG

57. Wotawa F (2020) Using model-based reasoning for self-adaptive control of smart battery systems. In: Sayed-Mouchaweh, M. (ed.) Artificial intelligence techniques for a scalable energy transition – advanced methods, Digital Technologies, Decision Support Tools, and Applications. Springer

58. Wotawa F, Stumptner M, Mayer W (2002) Model-based debugging or how to diagnose programs automatically. In: Proceedings of IEA/AIE’02, LNCS 2358, Springer-Verlag, pp 746–757

Publisher’s note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Franz Wotawa received an MSc and a Ph.D. from Vienna University of Technology in 1994 and 1996 respectively. He is currently a full professor of software engineering at the Graz University of Technology. His research interests include model-based and qualitative reasoning, theorem proving, mobile robots, verification and validation, and software testing and debugging.

David Kaufmann is currently a Ph.D. candidate at the Institute of Software Technology, Graz University of Technology, Austria. He works in the field of quality assurance methodologies for autonomous cyber-physical systems where the research focuses on reasoning techniques for validating fail-operational autonomous systems. David Kaufmann received his M.Sc. in Aviation from the FH Joanneum Graz, University of Applied Sciences.