Designing a Diagnostic Engine for Smart Environments

Eric Heiden[1], Sebastian Bader[2], and Thomas Kirste[2]

[1] University of Southern California
Los Angeles, USA
heiden@usc.edu
[2] Institute of Computer Science
Universität Rostock
Rostock, Germany
{sebastian.bader, thomas.kirste}@uni-rostock.de

Abstract. Automatically diagnosing a complex system containing heterogeneous hard- and software components is a challenging task. To analyze the problem, we first describe different scenarios a diagnostic engine might be confronted with. Based on those scenarios, a concept and an implementation of a semi-automatic diagnostic system are presented and some first benchmarks are shown.

Keywords: Automatic Diagnosis, Diagnostic Engine, Non-monotonic Reasoning, Model-based Diagnosis

1 INTRODUCTION

Imagine you enter a smart conference room, connect your laptop with the first available HDMI-port and the system automatically switches on the main projector. *Smart Environments* like this allow the user to seamlessly interact with an ensemble of interconnected devices. Sensors and actuators are combined to provide an unobtrusive environment in which the user's intentions are inferred to facilitate multimedia-enabled conferences or lectures. Immediately the first slide of your presentation appears on the screen and you can start your talk. But suddenly the projected display turns blank and you have to interrupt the presentation. What could possibly have happened? The green power indicating LED of the projector is still glowing and your laptop indicates it is duplicating its screen via the HDMI connection, too. Perhaps the display signal connection is broken? But after having manually checked the firmness of every cable on the way from your notebook to the projector the symptom still persists. Finally you see no other option than asking the facility manager to look after the problem. A short while later she finds the source of the error: the projector's lamp has exceeded its lifespan. Her expert knowledge helped her diagnose the problem.

This contrived scenario gives rise to several questions: How could this situation be handled better? What if we had expert knowledge immediately available

without always having to seek out technicians to identify and troubleshoot malfunctions? Which automatic methods do exist for the diagnosis of error sources?

In this paper, we present an approach for a diagnostic engine and its realization in our smart meeting room. Figure 1 shows the context of a diagnostic engine. The engine has predefined knowledge of the environment, i.e. the system of interconnected components to be diagnosed. Provided with a symptomatic observation, the engine reasons about possible sources of the encountered malfunction. To improve these explanations, further observations can be gathered using the middleware to retrieve status information from the system's components. As not every detail is observable through the middleware, the engine can also ask the user to provide observations.

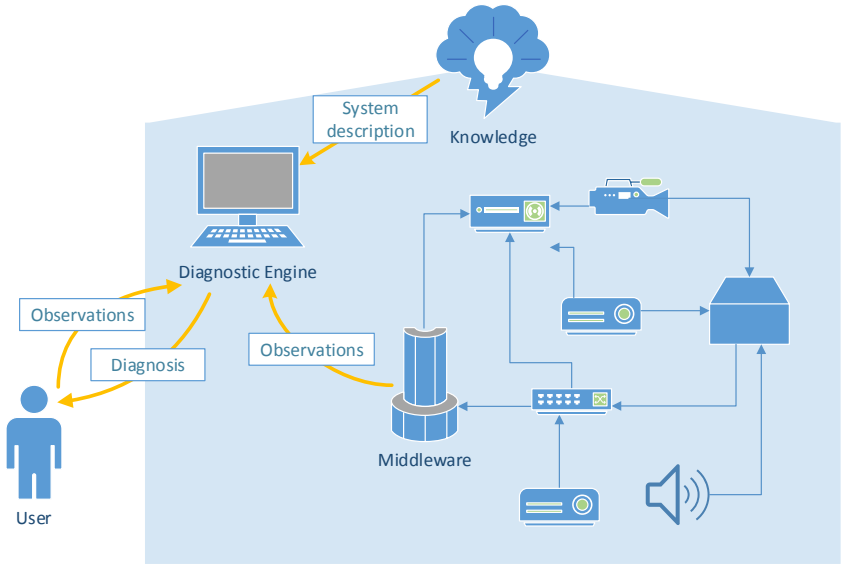

Fig. 1. Context of the Diagnostic Engine

First, we present a number of scenarios with increasing difficulty in Section 2. We review different approaches for diagnostic systems in Section 3. In Section 4, we present our concept for a diagnostic engine and describe our implementation in Section 5. Finally, we present some preliminary benchmark-results in Section 6.

2 DIAGNOSTIC SCENARIOS

In the following, several exemplary scenarios are described to demonstrate different aspects of failure diagnosis. Scenarios are grouped by their *difficulty* from a diagnostic engine's perspective ranging from problems which can be detected by a single automatic step, up to unsolvable issues which cannot be diagnosed, not

even in cooperation with the user. The analysis of problem classes furthermore allows to narrow down the diagnosis problem to only some subsystems which can be effectively treated by semi-automatic diagnosis.

Scenarios are described by first indicating what the diagnostic engine knows about the system. This a-priori knowledge consists of a wiring diagram representing the system description, and the user's observations on the state of the involved components. If we are confronted with a diagnosis problem, these observations should contain symptomatic descriptions which differ from the normal system behavior.

The *omniscient perspective* explains holistically the actual system configuration and true causes for the occurring issues. If the diagnostic engine were to know these facts, the proper diagnosis would be calculable instantaneously. Subsequently, the partial view of the system from the diagnostic engine's perspective is examined to derive all *possible diagnoses* the system can infer based on the observations and its understanding of the system. If the diagnosis is too vague to be useful for troubleshooting, the engine makes further observations or asks the user to perform these in order to gain more detailed information on the true causes of symptoms.

These scenarios are contrived and are meant to facilitate a more lucid view on the general idea of diagnostic reasoning and probable pitfalls of automatic diagnosing.

2.1 Automatically Identifiable Problems

This category contains scenarios which the diagnostic engine can identify completely autonomously without cooperating with the user, i.e. solely based on information accessible through the middleware.

Complete Observations The user tries to display content from laptop L on projector P but the screen remains blank. The user has already observed that P is switched off and that all involved devices and connections work as expected without any problems.

Omniscient Perspective:
- All components work correctly.
- All involved cables are in proper condition.
- The projector is switched off.

Possible Error Sources:
- The projector is switched off.

This scenario represents the special case where the provided observations are detailed enough to account for a precise diagnosis. No further knowledge is required to infer that the powered off projector is the single error source because it is given that all components function.

Automatic Observations The user tries to display content from laptop L on projector P but the screen remains blank. L and the control server S work as

expected, and L provides an HDMI video signal. All connections are stable. All devices are switched on.

Omniscient Perspective:
- All components work correctly.
- All involved cables are in proper condition.
- S configured P to use the wrong input port.

Possible Error Sources:
- P is defect (e.g. projector lamp burned out, firmware error, overheating, serious physical damage).
- S configured the wrong input on P.
- P *ignores* the configuration carried out by S and hence uses the wrong input.

The diagnostic engine is now confronted with an ambiguous situation where multiple diagnosis candidates compete. Here, a tie-breaking observation is necessary. Let us assume that the engine asks S which input configuration has been set on P. S now requests status information from P and reports P's input configuration. This automatic observation reveals that P is using the VGA port instead of HDMI to which L is connected to. From this knowledge it can be inferred that P is not defect (for simplicity we assume that if P can be accessed via Ethernet it is completely ok) and only S made the wrong input configuration. Without the need of human cooperation the engine could diagnose the symptom completely automatically. And the system could subsequently fix the problem automatically by reconfiguring P.

2.2 Semi-Automatically Identifiable Problems

Scenarios in which problems can only be diagnosed in cooperation with the user belong to this category.

Non-Automatically Observable System Properties Laptop L and projector P are switched on but P does not show any output. The HDMI connection between L and P is stable.

Omniscient Perspective:
- L works correctly.
- The HDMI connection between L and P is stable.
- The projector lamp in P has burned out.

Possible Error Sources: Let us assume that in this scenario we have a more detailed model of projector P than in the previous scenarios:
- P has a firmware bug.
- The projector lamp of P has burned out.
- The configured resolution of L is too high.

While the latter hypothesis can be discarded by performing an automatic test if L's resolution is greater than the maximum resolution supported by P, the ambiguity between the first two diagnosis candidates remains. In this case the user needs to check the projector-related properties.

It can be argued that those two observations can only be carried out by technicians and therefore the true cause of error is still hard to retrieve. On the other hand, the problem could be clearly limited to projector P by the diagnostic engine so that in a real world scenario the projector could just be replaced as an immediate retaliatory action.

Multiple Simultaneous Faults Both laptops, L_1 and L_2, the server S and the projector P are switched on and are in proper condition. However, P does not display anything.

Omniscient Perspective:
- All components work correctly.
- The HDMI cable between L_1 and P, and the VGA cable between L_2 and P are broken.
- P is configured to use the VGA port as input source.

Possible Error Sources:
- The HDMI cable between L_1 and P is broken.
- The VGA cable between L_2 and P is broken.
- The HDMI cable between L_1 and P and the VGA cable between L_2 and P are broken.

First, the diagnostic engine should determine which input has been selected by P. Therefore, S is commissioned to request information on the input configuration of P. This returns VGA as the active input port and thus it can be reasoned that the VGA connection between L_2 and P is broken.

Now that we are certain about one component of our diagnosis it would also be helpful to know if there might be any more faults in the system. Let us now assume that P's input configuration is read-only so that S cannot alter the setting automatically. However, in order to check whether the HDMI connection between L_2 and P is broken, we need to test if P also does not show anything if HDMI is selected as input. Thus, the diagnostic engine has to ask the user to carry out this diagnostic action and then tell whether P shows something or not. The user reports that the screen is still blank after HDMI has been defined as input. Ultimately, the diagnostic engine concludes the following: both the HDMI connection between L_2 and P as well as the HDMI connection between L_1 and P are broken.

If the user was unable to perform the diagnostic step of altering the system's configuration to provide a new observation, it would not be decidable whether the HDMI connection between L_1 and P was broken or not. A common approach to handle this knowledge gap would be to assume that everything is working correctly unless we have concrete evidence to retract such assumption (e.g. by observing a symptom). Below we shall see how this type of so-called *non-monotonic* reasoning is handled from a logical perspective.

Disconnected Subsystems Until now, we have assumed that our diagnostic engine somehow can instruct system-inherent components, e.g. servers, to perform observations or alter the system configuration. In reality however, the

diagnostician itself is a device or a software running on network-attached hosts which might fail or loose connection – just like any other component. To examine this property in detail, let us assume the following scenario: Laptop L, the diagnostic engine D und the NAS (Network-Attached Storage) are connected to switch S via Ethernet. While all devices are working as expected, L cannot connect to NAS.

Omniscient Perspective:
- All components work correctly.
- The Ethernet between D and S is broken.
- The Ethernet connection between NAS and S is broken.

Possible Error Sources:
- The Ethernet connections $D - S$ and $NAS - S$ are broken.
- The Ethernet connections $D - S$ and $L - S$ are broken.
- The Ethernet connections $D - S$, $L - S$ and $NAS - S$ are broken.

The diagnostic engine cannot perform further observations automatically because it is disconnected from the subsystem $B = \{NAS,\ L,\ S\}$. Therefore, all non-empty permutations of broken Ethernet connections between components $c_1, c_2 \in B$, together with the fact that Ethernet connection $D - S$ is broken, are valid diagnoses. This ambiguity can only be resolved if D gets access to S or by asking the user to check all connections.

2.3 Unidentifiable Problems

This class comprises diagnostic scenarios where the diagnostic engine is unable to diagnose the symptoms – not even in cooperation with a human user.

Hidden Interactions After switching on a lamp, all lamps and projectors are suddenly powered off.

Omniscient Perspective:
- All components work correctly.
- The lamps and projectors are connected to power sources which share the same fuse.
- Due to power overload of too many connected devices, this fuse was tripped and caused the power outage for all projectors and lamps.

Possible Error Sources: N/A

The engine cannot propose any diagnoses since it is not aware of the interaction between projectors and lamps. The fact that their power sources share the same fuse might not be indicated in the wiring plan. The engine cannot even ask the user to check the fuses because it can only reason about facts which are provided as system knowledge.

Several approaches exist to handle hidden interactions (see [2, 9]). If we observe a malfunction which affects completely unrelated parts of the system like in the given scenario, it can be assumed that hidden, unintended interactions have occurred.

Intermittent Abnormalities In this scenario, the diagnostic engine runs directly on the user's laptop L.

The switch S intermittently becomes unavailable. Laptop L is working correctly and the Ethernet connection between L and S is stable.

Omniscient Perspective:
- Laptop L is working correctly.
- The Ethernet connection from L to S is stable.
- Switch S is congested sometimes when the user wants to access it.

Possible Error Sources: N/A

Imagine that coincidentally, every time the diagnostician tries to access S, the switch responds immediately. This observation contradicts with the user's observation and in contrast shows no symptoms. Therefore, it can only be reasoned that S is working correctly despite the apparent malfunction.

This problem could be addressed by letting the diagnostic engine constantly observe every network access L is making. This kind of *online* diagnosis would then experience – just like the user – the symptomatic timeouts. However, during the course of this paper only offline scenarios are considered in which a malfunction has occurred and the user subsequently requests an explanation of the observed symptoms.

Wrong Observations Laptop L and projector P are switched on but P does not show any output (cf. Scenario 2.1).

Omniscient Perspective:
- All components work correctly.
- The projector is configured to select its *HDMI* port as input source.
- The observation provided by the user is wrong, i.e. P indeed displays content from L.

Possible Error Sources:
- P is broken.

This diagnosis (that P is broken) is wrong because the diagnostician trusted the user-provided observation and does not have any automatic verification methods for given observations. Observations (human-made or automatic) do not need to be invalid only because of human deceit, often technical devices themselves report false status information if they are broken, i.e. they are behaving *abnormally*. This problem could be tackled by assigning quantifying the degree of belief when dealing with arbitrary inputs. A diagnostic theory of involving probabilistic reasoning to deal with uncertainty is given by [11].

3 PRELIMINARIES

Before introducing two main concepts of diagnostic reasoning, namely *consistency-based* and *abductive* diagnosis, this section covers the general notion of diagnosis in the real-world context and its development to an A.I. discipline.

While the term *diagnosis* can also mean the pure decision whether a system is working or not, this paper is based on the definition of diagnosis as a method to identify the causes of observed system faults as precisely as possible.

Similar to real world diagnosis where experts are asked to find those parts in complex systems, like cars or powerhouses, which account for the observed malfunction, *diagnosis* as a subfield in artificial intelligence similarly provides techniques to identify causes of observed symptoms – especially for applications which require significant expert knowledge. This task requires observations of the actual, possibly unexpected system behavior as well as knowledge of the problem domain sufficient enough to infer meaningful conclusions.

Diagnostic Engines first emerged during the late 1960's to early 1970's in the form of *rule-based expert systems* [1]. Causal representations of symptoms and faults were explicitly written as hard-coded or compiled knowledge as an attempt to mimic human expert behavior. These systems had major drawbacks when applied to non-static domains were properties evolve and affect the causal relationships between observed problems and underlying error sources. Knowledge engineers were required to cooperate with human experts in order to manually update the knowledge base.

In contrast to these heuristic approaches, *model-based diagnosis* which emerged during the 1980's, allowed for an estimation of system behavior which can be compared to the observed outcomes in order to detect abnormalities. These systems showed a higher degree of robustness compared to rule-based systems because they could better handle unexpected cases.

Current trends in diagnostic systems present the coupling of classical model-based diagnosis with other AI techniques like neural networks or genetic algorithms [1] to improve knowledge acquisition and diagnose complex and dynamic systems more effectively.

Model-based diagnosis is a commonly used framework that works by modelling a system consisting of interacting components or subsystems via logical formulas. While it possible to define fault models, during the course of this paper only the system's *expected* behavior is modelled as in [14, 4, 11]. Conversely, the outputs of the real-world implementation of the system are measured and the observations are as well logically formalized. The discrepancies between the observations and the predicted behavior are finally used to diagnose faulty components whose behavior contradict the model's behavior.

Reiter proposed in [13] a logic for default reasoning called *default logic* which extends first-order logic by allowing to perform default assumptions. While in standard logic it can only be stated that something is either TRUE or FALSE, default logic can express facts that are typically TRUE only with a few exceptions.

For example, the fact that almost all projectors have a VGA port can be represented as follows in default logic:

$$\frac{\textsc{projector}(x) : \mathrm{M}\ \textsc{has-vga}(x)}{\textsc{has-vga}(x)}$$

Here M stands for it is consistent to assume so that this default rule can be read as: given the fact that x is a projector (prerequisite) and it is consistent to

believe that x has a VGA port (justification), then one may assume that x has a VGA port (conclusion). To specify the notion of this consistency requirement, the semantics of default logic is described in the following.

A default theory is defined as a pair (W, D), where D is a set of default rules and W is the set of logical formulas which define our background theory. If the prerequisite of a given default D is *entailed* from our theory W and every justification is *consistent with* W than we can add the conclusion to the theory.

Since the consequence relation is not monotonic, default reasoning is a kind of *non-monotonic reasoning*.

First-order logic is *monotonic*, i.e. given two sets A and B of first-order formulas where $A \vdash w$, then every model of $A \cup B$ is also a model of A so that $A \cup B \vdash w$. If we assume B to be newly discovered information, the addition of B to our existing knowledge A does not affect the outcome w of $A \cup B$ with respect to the models of A. That means, if later discoveries reveal contradictions to formerly assumed rules they cannot be retracted.

However, in any reasoning method were assumptions or beliefs are made, like default reasoning or abductive reasoning (Section 3.2), it is necessary to retract an assumption in order to avoid inconsistencies with newly gained evidence. Adding knowledge to the theory which contradicts the assumptions shall invalidate them and thus reduce the set of conclusions that can be derived from the theory. The notion that further evidence does not monotonically grow the set of derivable propositions describes the property of *non-monotonic reasoning*.

3.1 Consistency-Based Diagnosis

The theory by Reiter on *Diagnosis from First Principles* [14] is a model-based approach which conjectures about faulty components by only selecting hypotheses which are consistent with the system's model and the observations. This approach laid an important foundation for the automatic identification of problems not only in electric circuitry as in the early days of automatic diagnosis, but universally across many different domains. In the following, diagnosis from first principles will be used to present the concept of *consistency-based diagnosis*.

In this model-based approach the only information available to explain discrepancies between the observed and correct system are *first principles*, i.e. the model of the *system*'s expected behavior represented using logical formulas. Reiter's theory is applicable to any logic $\mathcal{L}$ which fulfills the following criteria:

1. *Binary semantics* so that every sentence of $\mathcal{L}$ has value TRUE ($\top$) or FALSE ($\bot$).
2. $\{\wedge, \vee, \neg\}$ are supported logical operators which have their usual semantics in $\mathcal{L}$.
3. $\models$ denotes semantic entailment in $\mathcal{L}$.
4. A sound, complete and decidable theorem prover exists for $\mathcal{L}$.

In general, first-order logic (FOL) is only *semidecidable*, i.e. the question whether an arbitrary formula f is logically valid (a theorem) in $\mathcal{L}$ can always be

answered correctly whereas a negative or no answer at all will be given if f is not valid in $\mathcal{L}$. To still fulfill the last criterion, we will from now on define a decidable subset of FOL as the logic $\mathcal{L}$ to be used for diagnosis. This is established by requiring $\mathcal{L}$ to be a FOL with finite domain of discourse D (*Herbrand universe*) and finite *Herbrand base*.

Definition 1 (System). *The* system *is defined as a pair* (*SD*, *COMPONENTS*) *consisting of SD, the* system description *which contains rules of logic $\mathcal{L}$ describing the system's normal behavior, and COMPONENTS, the finite set of constants representing the* components.

Components can be devices, connections, subsystems, or any entity that could be (partially) responsible for the system's malfunction and should therefore be included in the diagnosis. The system description makes use of AB(c)-predicates which express for component $c \in$ COMPONENTS that c is behaving "abnormally". Thus, when modelling the intended system behavior these abnormal-predicates only occur in its negated form.

For a running example let us revisit our first fully automatically diagnosable scenario. In addition to the system properties described before, let us also assume that projector P's lamp B is a component which can burn out preventing P to show anything.

```
works(l) :- not ab(l).
works(p) :- not ab(p).
shows_image(p) :- works(l), works(p),
                  not ab(b), not ab(hdmi).
```

Fig. 2. Prolog implementation of a simple diagnosis example

We could represent this scenario using the following system description SDas a set of definite Horn clauses shown in Figure 2 Similarly to SD, the *observations* are given as a finite set of logical formulas in L as well. A *diagnostic problem* is defined as the triple (SD, COMPONENTS, OBS). For this example, let OBS $= \{\neg$ _IMAGE(P), WORKS$(L)\}$ be the set of our observations, i.e. we observed that P did not show an image and L was working.

A *diagnosis* for the problem (SD, COMPONENTS, OBS) is defined as the *minimal* set (under set inclusion) $\Delta \subseteq$ COMPONENTS where SD $\cup$ OBS $\cup \{$AB$(c) \mid c \in \Delta\} \cup \{\neg$AB$(c) \mid c \in$ COMPONENTS$\backslash\Delta\}$ is consistent.

Generally, a valid diagnosis would be the trivial solution that all components are faulty, since we are following the model-based approach were only the expected behavior is known. Therefore, the *Principle of Parsimony* has been established in [14] and advocates the *minimal diagnosis*. To find minimal diagnoses it helps to reformulate the aforementioned definition of Δ in terms of *conflict sets*.

A *conflict set* for $(\texttt{SD}, \texttt{COMPONENTS}, \texttt{OBS})$ is defined as $C = \{c_1, \ldots, c_2\} \subseteq \texttt{COMPONENTS}$ such that $\texttt{SD} \cup \texttt{OBS} \cup \{\neg\text{AB}(c_1), \ldots, \neg\text{AB}(c_n)\}$ is *inconsistent*. A conflict set C is *minimal* iff no subset $C' \subset C$ exists that is also a proper conflict set satisfying the equation.

Consistency-based diagnosis interprets the consistency requirement by the semantics of classical logic where a logical formula $f \in \mathcal{L}$ is *consistent* if f has *model*, i.e. an *interpretation* or assignment of variables of f to the domain of discourse D so that the meaning of f is TRUE. Therefore, we can assume the unresolved consistency terms $\text{WORKS}(P)$ and $\neg\text{WORKS}(P)$ to be TRUE since a model exists for $\texttt{SD} \cup \texttt{OBS} \cup \{\text{AB}(c) \mid c \in \Delta\} \cup \{\neg\text{AB}(c) \mid c \in \texttt{COMPONENTS} \backslash \Delta\} \cup \{\text{WORKS}(P)\}$ and another model exists for $\texttt{SD} \cup \texttt{OBS} \cup \{\text{AB}(c) \mid c \in \Delta\} \cup \{\neg\text{AB}(c) \mid c \in \texttt{COMPONENTS} \backslash \Delta\} \cup \{\neg\text{WORKS}(P)\}$. As we will later see, this consistency definition of non-stable models marks the fundamental difference to *abductive reasoning* (cf. Section 3.2). The following two conflict sets can be found:

$$C_1 = \{\text{AB}(B), \text{AB}(P), \text{AB}(HDMI)\}$$
$$C_2 = \{\text{AB}(B), \text{AB}(L), \text{AB}(P), \text{AB}(HDMI)\}$$

Hence we have the minimal conflict set is $C_1 = \{\text{AB}(B), \text{AB}(P), \text{AB}(HDMI)\}$.

For a collection S of sets, a *hitting set* H for C is a set $H \subseteq \bigcup_{S \in C}$ such that $\forall S \in C : H \cap S \neq \emptyset$. A hitting set is *minimal* if no proper subset of it is a hitting set. Reiter uses this definition to reformulate the characterization of diagnoses: $\Delta \subseteq \texttt{COMPONENTS}$ is a *diagnosis* for $(\texttt{SD}, \texttt{COMPONENTS}, \texttt{OBS})$ iff Δ is a *minimal hitting set* for the collection of *minimal conflict sets* for $(\texttt{SD}, \texttt{COMPONENTS}, \texttt{OBS})$. For our example, three minimal hitting sets can be found which represent our minimal diagnoses $\Delta_1 = \{\text{AB}(B)\}, \Delta_2 = \{\text{AB}(P)\}, \Delta_3 = \{\text{AB}(HDMI)\}$.

3.2 Abductive Diagnostic Reasoning

Abductive reasoning is a form of logical inferencing that hypothesizes explanations for a given observation, and is viewed as a competing concept to consistency-based diagnosis. As a powerful concept to handle commonsense reasoning, it has been applied in the diagnosis domain [5].

Abduction became a powerful reasoning method to Artificial Intelligence especially in the field of diagnosis which is considered by [3] as one of the most representative and best understood application domains for abductive reasoning. It has further served as a basis for other types of expert systems, e.g. in the medical domain, and apart from diagnosis in areas such as planning, natural language understanding and machine learning [3].

Abduction is a logical reasoning method that generates, given a logical theory or domain knowledge T and a set of observations O, explanations (= hypotheses) E which explain O according to T such that $T \cup E \vDash O$, and $T \cup E$ is consistent. Abductive reasoning is a type of *non-monotonic reasoning* since hypotheses E which have been made given theory T and observations O might become obsolete due to new observations O' which require the reasoning system to retract those explanations which do not meet the two constraints from above. Therefore,

default reasoning can be based on abduction instead of non-monotonic logics so that defaults are represented as *hypotheses* to be made or retracted instead of deriving *conclusions* within non-monotonic logics (cf. [6]).

An *abductive theory* is a triple (P, IC, A), where P is a logic program defining the domain knowledge, IC is a set of integrity constraints (logical formulas) which define constraints on the abduced predicates, and A is a set of abducible ground atoms.

We can now define express (P, IC, A) in terms of the diagnosis domain in order to identify faulty components $\Delta \subseteq \texttt{COMPONENTS}$ in a malfunctioning system in the same way as finding the best explanations for given symptoms. A *system* $(SD, COMPS)$ is formalized as follows: SD is the system definition, as defined by P, and COMPS is the set of system components which can be possible sources of errors, as defined by A.

The integrity constraints IC can be used to additionally constrain the generated diagnose, e.g. by stating that certain components $A' \subseteq A$ cannot be diagnosed as faulty.

When diagnosing a system, one needs to observe the malfunction and represent these symptomatic observations as a set of logical formulas OBS. The diagnosis problem $(\texttt{SD}, \texttt{COMPS}, \texttt{OBS})$ is solved through abduction by retracting some of the $\neg$AB-assumptions. The resulting set $\Delta \subseteq A$ is a valid diagnosis if it explains all of the observed symptoms.

To meet the goal of providing useful diagnoses which do not contain any, for the fault explanation insignificant components, the *Principle of Parsimony* advocates *minimal diagnoses*. Hence, a diagnosis for $(\texttt{SD}, \texttt{COMPS}, \texttt{OBS})$ is according to [12] a minimal set $\Delta \subseteq A$ such that $\texttt{SD} \cup \Delta \models \texttt{OBS} \cap IC$.

We will use implementations of abductive reasoning in the form of logic programming and answer set programming. These systems follow the *stable model semantics* which was motivated by formalizing the behavior of SLDNF resolution (selective, linear, definite resolution with negation as failure), a common resolution strategy for logic programming systems like Prolog.

For any set M of atoms from Π, let Π_m be the program (reduct) generated from Π by removing

1. each rule that has a negative literal $\neg l$ in its body where $l \in M$, and
2. all negative literals in the bodies of the remaining rules.

Since Π_M is now negation-free, it has a unique minimal Herbrand model. If this model is equal to M, then M is a *stable set* of Π [7].

Answer Set Programming (ASP) is a form of declarative programming which is primarily addressed to solving NP-hard search problems. It has its roots in Reiter's theory of *default reasoning* and in the generation of *stable models*.

In ASP, search problems are first ground-instantiated by so-called grounders like LPARSE which are front-ends accepting logic programs. In the next step, ASP solver like SMODELS or DLV solve these computable search problems by calculating all stable models of the grounded programs. Unlike SLDNF-employing reasoning tools like Prolog, ASP solvers always terminate [10]. In

addition, the performance of current ASP solvers is comparable to highly efficient SAT solvers because similar algorithms are used.

Consistency-based and abductive diagnosis both represent techniques for identifying the error sources of a malfunctioning system. Although these methods can be applied to the same task, the results that are calculated sometimes differ. In contrast, abductive diagnosis is more restrictive on the selection of diagnostic explanations: the diagnosis Δ in conjunction with the system description `SD` must have a *stable model* which logically *entails* the observations.

One difference between consistency-based and abductive diagnosis is that the former applies a weaker criterion on valid diagnoses, because it uses the consistency formula in the traditional FOL semantic.

4 CONCEPT

The following section presents the conceptual ideas and algorithm behind the implemented diagnostic engine.

4.1 Refining Hypotheses

The diagnosis Δ is a set of hypotheses which explain the system's malfunction based on the given knowledge as logic program P and observations `OBS`. Although can already calculate minimal diagnoses which only select as few components as possible using consistency-based or abductive reasoning, there are often too many explanations given to efficiently isolate the true causes of the problem. In fact, model-based diagnosis is often criticized for not being able to 'pinpoint a failing component from the available symptom information' [8].

Therefore, further observations are necessary to refine the diagnosis. Let us assume that P is given by the set of Horn clauses and our logic program supports *negation as failure* to be interpretable within *stable model semantics*.

What could be a further observation? Consider our running example and its system description shown as logic program in Figure 2. In the context of semi-automatic diagnosis it is assumed that the user does not know the true source of errors, i.e. faulty components represented by AB-predicates. Based on the observation of `not shows_image(p)` we can only propose to observe `works(l)` or `works(p)` as these two predicates belong to rules in P which further contain AB-predicates. After the initial observation of `not shows_image(p)` the set of minimal diagnoses would be $\{\{l\},\{p\},\{b\},\{hdmi\}\}$. If the user would observe `works(l)`, then according to stable model semantics the diagnostic engine would retract $\{l\}$ from the diagnosis. This refinement of the diagnosis exemplifies non-monotonic reasoning where additional knowledge leads to the retraction of assumptions.

Starting from the initial, non-empty diagnosis Δ_0 which has been computed by at least a single observation of the system's symptoms (otherwise $\Delta_0 = \emptyset$ since P is consistent) the diagnostic engine proposes predicates $G \subseteq P \backslash (\texttt{OBS} \cup \{\neg l \mid l \in \texttt{OBS}\})$ which have not yet been observed (neither negated nor positive). If the

user or an automatic middleware system can observe the predicate $p \in G$, p or $\neg p$ is added to our observations `OBS`, depending on what was observed about p. If p or $\neg p$ are not observable, the diagnostic engine should propose a new predicate for observation, if available. Otherwise, no more observations can be proposed.

Algorithm 1 FindAbducibles$(p, T, \texttt{OBS})$ calculates the set of abducibles $\subseteq$ `ABDUCIBLES` which can be abduced from $T \cup \{p\}$ in case p is observed. Here, a reasoning system, e.g. ASP system is required in order to calculate the stable models of the current theory.

Algorithm 1 Finding abducibles

```
procedure FindAbducibles(p, T, OBS)
    R ← ∅
     // CSM = CalculateStableModels
    for all A ← CSM(T ∪ OBS{p}, ABDUCIBLES) do
        R ← R ∪ A
    end for
    return R
end procedure
```

Note that we are unifying the minimal diagnoses which possibly consist of multiply components which together must be faulty in order to explain the given symptom. Instead of having sets of sets of possibly faulty entities, the set representation of all candidates allows us to quantify for each diagnosis step the utility of an observation.

4.2 Proposing Observations

The information which abducibles can be eliminated from the diagnosis Δ if we observe that a predicate p or its negation is true can then be used to propose such p which maximally reduces Δ. Thanks to the Algorithm 1 we can calculate which diagnosis (or any) would result from observing p or $\neg p$ so that we can select a p which would result in the smallest possible diagnosis $\neq \emptyset$.

4.3 Interactive Diagnosis

The diagnosis process as defined in Algorithm 3 starts with the knowledge base, i.e. program, P and a symptomatic observation `OBS` which is a set of positive or negated predicates occurring in P. The diagnosis Δ is calculated by accumulating all abducibles $\in A = \{\text{AB}(\dots)\}$ which explain the given observation `OBS`. Using further, proposed observations as from Algorithm 2, the diagnosis is refined or assured: Abducibles contradicting Δ will be used to reduce Δ by removing the negations of these abducibles. Abducibles which confirm Δ will be added to the fixed diagnosis Δ_f. Δ_f is reflected during the diagnostic reasoning process

Algorithm 2 Selecting the optimal observation.

Require: $\Delta \neq \emptyset \wedge \texttt{OBS} \neq \emptyset \wedge \Delta_f \subset \Delta \wedge \forall p \in P : \textsc{ObservingCost}(p) > 0$
  procedure PROPOSEOBSERVATION($P, \Delta, \Delta_f, \texttt{OBS}$)
    $O \leftarrow \emptyset$
    for all $p \in P \backslash (\texttt{OBS} \cup \{\neg l \mid l \in \texttt{OBS}\})$ **do**
      $p_{\text{AB}+} \leftarrow \textsc{FindAbducibles}(p, P, \texttt{OBS})$
      $p_{\text{AB}-} \leftarrow \textsc{FindAbducibles}(\neg p, P, \texttt{OBS})$
      $v \leftarrow \min\{p_{\text{AB}+}, p_{\text{AB}-}\}$
      // if abducibles can be calculated.
      $O[p] \leftarrow v$
    end for
    if $O \neq \emptyset$ **then**
      return $\arg\min_{p \in P \backslash \texttt{OBS}} O[p]$
    else
      return $\perp$
    end if
  end procedure

using the *Integrity Constraints IC*. These constraints limit the calculated set of abducibles by only allowing abducibles which do not conflict with Δ_f. The diagnosis finishes if no further observation can be proposed or if Δ is small enough so that the user can troubleshoot the problem.

5 IMPLEMENTATION

The implemented diagnostic system covers the full workflow from extracting knowledge of semistructured wiring information to interactively providing the user with diagnoses. This section first covers the general architecture of the implemented system and then describes the necessary implementation steps in detail from start to finish of the diagnosis workflow.

5.1 Architecture

The module for *knowledge extraction* takes as input wiring information from a CSV (comma-separated values) table and is further provided with device-related code to model the behavior of different devices as well as the interaction between them. This knowledge can be represented via disjunctive datalog and abductive logic programs. By formulizing the wiring graph in DOT (a graph description language) it can be visualized by graph drawing tools like *dot* or *neato* from the *Graphviz* package.

The formal description as a logic program can then be used via *ProLogICA* to find diagnoses by performing abductive reasoning or *DLV* to diagnose using answer set programming. The implemented *Diagnostic Engine* uses DLV to generate diagnoses if provided with an initial observation by the user. An interactive

Algorithm 3 The complete diagnosis process.

Require: $P \neq \emptyset \wedge \texttt{OBS} \neq \emptyset$

Require: $\textsc{Abduce}(G, P, IC, A)$ queries the Abductive Reasoner given the theory (P, IC, A) and goal G, and subsequently yields all minimal solutions $\subseteq \mathcal{P}(A)$ which explain G.

procedure $\textsc{Diagnose}(P, \texttt{OBS})$
  $\Delta \leftarrow \emptyset$
  $\Delta_f \leftarrow \emptyset$
  while (Δ not refined enough and
   $\exists p \in P : p \notin \texttt{OBS}$ and $\neg p \notin \texttt{OBS}$ and
   (($\{p\} \cup \texttt{OBS} \cup P \cup \Delta_f$ consistent) or
    ($\{\neg p\} \cup \texttt{OBS} \cup P \cup \Delta_f$ consistent))) **do**
    $\Delta \leftarrow \bigcup \textsc{Abduce}(\texttt{OBS}, P, \Delta_f, \{\textsc{ab}(\dots)\})$
    $\beta \leftarrow \textsc{ProposeObservation}(P, \Delta, \Delta_f, \texttt{OBS})$
    if $\beta = \bot$ **then**
      return Δ
                                ▷ No more observations can be proposed
    end if
    if β is observable **then**
      $(p_{\textsc{ab}+}, p_{\textsc{ab}-}) \leftarrow$
        $\textsc{FindAbducibles}(\beta, P, \Delta, \Delta_f, \texttt{OBS})$
      if β is observed as *false* **then**
        $\texttt{OBS} \leftarrow \texttt{OBS} \cup \{\neg\beta\}$
        $\Delta \leftarrow \Delta \backslash \{\neg l \mid l \in p_{\textsc{ab}-}\}$
        $\Delta_f \leftarrow \Delta_f \cup p_{\textsc{ab}-}$
      else
        $\texttt{OBS} \leftarrow \texttt{OBS} \cup \{\beta\}$
        $\Delta \leftarrow \Delta \backslash \{\neg l \mid l \in p_{\textsc{ab}+}\}$
        $\Delta_f \leftarrow \Delta_f \cup p_{\textsc{ab}+}$
      end if
    end if
  end while
  return Δ
end procedure

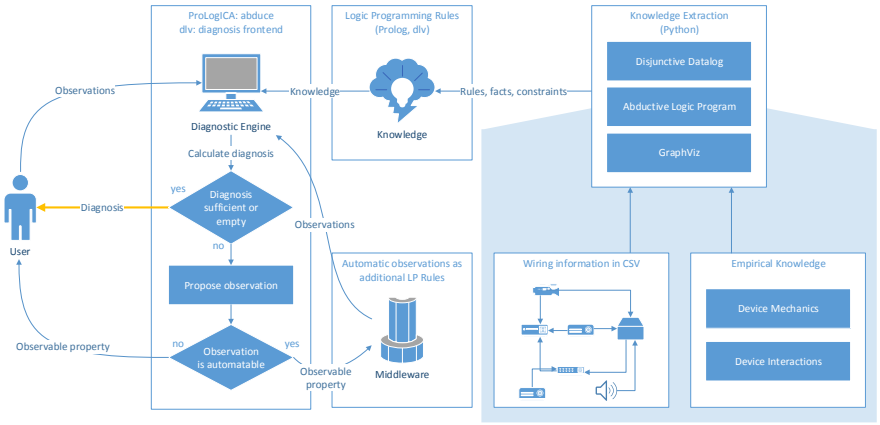

Fig. 3. Architecture of the implemented Diagnostic System

loop has been implemented to refine the calculated diagnosis based on automatic observations by querying the middleware (e.g. publish-subscribe infrastructure) or by instructing the user to conduct observations of non-automatically observable system properties.

The interactive diagnosis session finishes once the user accepts the diagnosis as refined enough to troubleshoot the symptoms, or if there are no further observable (neither by a human nor the middleware) system properties which could improve the diagnosis.

5.2 Knowledge Representation and Reasoning

Diagnostic reasoning heavily depends on the provided information of the implemented system and thus can be seen as a discipline belonging to *knowledge representation and reasoning* (KR).

The logical representation of a system and its diagnosis using logical reasoning has several advantages over other diagnosis approaches:

- Logical formulae to describe the system structure can be extracted easily from existing system data, e.g. wiring diagrams or technical manuals.
- The logical rules are human-interpretable so the user can comprehend and, if necessary, reproduce the diagnostic reasoning.
- Only normal behavior needs to be modeled allowing for smaller (human) effort to define the system since no fault definitions are required.
- It is ensured that the calculated diagnosis is *minimal*, i.e. the minimal set of possible sources of error is returned. This reduces further troubleshooting efforts.

Because of the pecularities of the two target platforms, ProLogICA and DLV, two separate modules have been implemented to formalize wiring information as

logic programs, namely Abductive Logic and Datalog Programs. This decision was enforced for the following reasons:

- ProLogICA relies on the non-declarative (linear) semantic of python so that transitive connections cannot be defined recursively (in contrast to DLV) as

$$conn(A, C) : -conn(A, B), conn(B, C).$$

  but instead must be stated using a helping predicate rconn which handles the recursion so that

$$rconn(A, C) : -conn(A, B), rconn(B, C).$$

- ProLogICA exhibits poor performance on too many nested and especially non-grounded rules. Thus, connections between all devices have been resolved via depth-first search.
- Not-AB-statements cannot be written as *not(ab(...))* but only *not ab(...)* in DLV while the latter syntax was not supported on the used SWI Prolog implementation.
- DLV requires AB-statements to be grounded, while this leads to cryptic constants in Prolog.

5.3 Available Implementations

The diagnostic engine has been realized using two different implementations: the answer set programming environment DLV and the abductive reasoning tool ProLogICA.

DLV stands for **D**ata**L**og with Disjunction (where V represents the logical operator $\vee$) and is a disjunctive logic programming system. Rules can be written in disjunctive datalog (function-free) of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_{k+m}.$$

which allows DLV as an ASP system to solve problems whose complexity lies beyond the solvable scope of non-disjunctive programming. DLV imposes a safety condition on variables in rules such that a rule is logically equivalent of its Herbrand instances.

DLV provides a diagnosis front-end for Abductive Diagnostic Reasoning as well as for Consistency-Based Diagnosis. As described in [5], a diagnostic problem represented by the set of observations `OBS`, the system description `SD` and the set of AB-atoms can be rewritten in disjunctive datalog so that every stable model which the ASP solver finds represents a diagnosis. The input programming language for this front-end however does not support disjunctive datalog and instead falls back to traditional datalog (function-free logic programming).

ProLogICA is an implementation of Abductive Logic Programming (ALP) in Prolog. It allows the user to define in a single file the abductive theory (P, IC, A), where P represents the set of rules to describe the domain knowledge, IC is a set of integrity constraints, and A declares the abducible predicates [12].

In contrast to competing implementations of abductive reasoning, ProLogICA allows the occurrence of negated abducibles so that the formalization of normal system behavior can be made as described in Reiter's Theory.

5.4 Semi-Automatic Diagnosis

Human-machine cooperation is realized via proposing system properties to the user which would improve the diagnosis. This section presents the implementation of this fundamental aspect of *semi*-automatic diagnosis.

Given a scenario as logic program which represents the system description, and a file of hypothesis declarations which describe the possible AB-predicates to be assumed as a diagnosis, the user first needs to provide an initial observation. Then, a diagnosis is calculated. If this calculation fails, the observations contain a contradiction or no hypotheses could be found. In order to simulate automatic observations, an additional file can be provided which contains non-abducible predicates either negated (using `not`) or non-negated. If the ranking of possible observations returns predicates that are automatically observable, these observations are added to the set of current assumptions. If there are no automatic observations, the user is asked to perform a proposed observation. These observations help to refine the diagnosis so that $|\Delta_{n+1}| \leq |\Delta|$ for every loop of the interactive diagnosis.

6 RESULTS

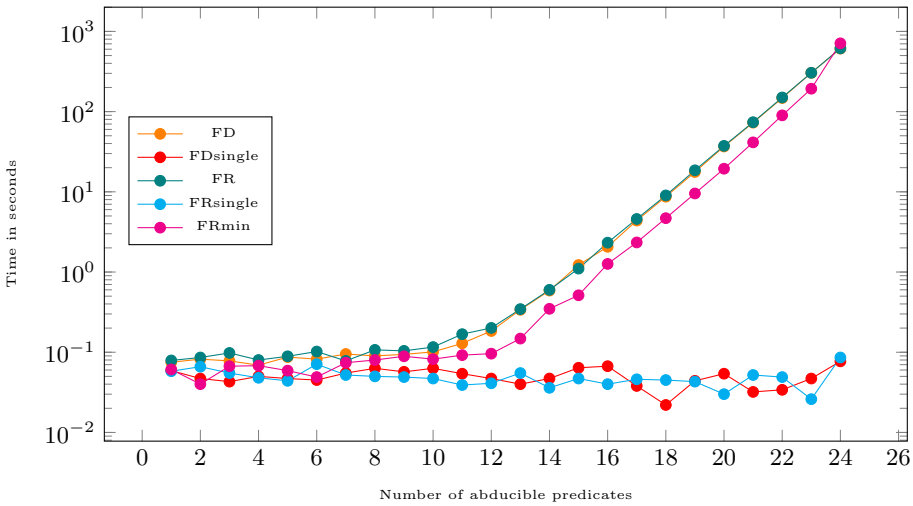

Fig. 4. Benchmark Results of the DLV Diagnosis Frontend

The results show that it is unfeasible for DLV to calculate minimal diagnoses under the subset-relation. Even for 24 possible faults in the given scenario, it took more than 12 minutes to calculate a diagnosis. However, single fault diagnosis and non-minimal diagnosis showed the vast performance gain which ASP systems can provide. Thanks to grounding of the given program and an efficient solver, DLV is able to handle complex scenarios and logic programs with (in our case more than 1500 lines of code) efficiently.

ProLogICA did exhibit no such problems as DLV when calculating minimal diagnoses. Although the number of abducible predicates does not seem to considerably influence its calculation performance, the type of knowledge representation played a crucial role whether ProLogICA was able to find a solution, or to not terminate. Especially rules which were highly nested, or with non-grounded variables constantly inhibited ProLogICA from terminating or finding useful solutions.

7 SUMMARY

The implemented diagnostic engine provides an interactive environment where in cooperation with the user a sufficient diagnosis can be found. The paper provided a theoretical background, discussed several approaches and the algorithmic framework to realize semi-automatic diagnosis in smart environments.

However, several qualifications must be imposed in order to guarantee useful and timely diagnoses. With the current implementation, either single faults can be detected efficiently despite a complex system description, or the system's model needs to simplified. This can be done by avoiding deep nesting in the system description or by limiting the set of possible fault candidates.

The diagnostic engine could be improved by further implementing context knowledge of the components to be diagnosed. Heuristics could be applied to automatically limit the set of faulty components. If the model-based diagnostic engine would have empirical information available to not treat every component equally as a potential source of malfunction, the selection of diagnosis candidates could be greatly accelerated. Heuristic knowledge would also provide the user with better explanations from the beginning since empirical information on probable faults can be used. These *symptom-failure association rules* could be learned from experience (cf. [8]) to better mimic the expertise of a human diagnostician. Performance improvements could be made over pure model-based diagnosis due to the caching of rules.

References

1. C Angeli. Diagnostic Expert Systems: From Experts Knowledge to Real-Time Systems. *Advanced Knowledge Based Systems: Model, Applications & Research*, 1:50–73, 2010.
2. Claudia Böttcher. No Faults in Structure? How to Diagnose Hidden Interaction. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1728–1735, 1995.

3. Henning Christiansen. Abductive reasoning in Prolog and CHR. *Science*, pages 1–18, 2005.
4. Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.
5. Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42(1):3–42, 1995.
6. K Eshghi and R a Kowalski. Abduction Compared with Negation by Failure. *Proceedings of the Sixth International Conference on Logic Programming*, (JANUARY):234–254, 1989.
7. M. Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *5th International Conf. of Symp. on Logic Programming*, pages 1070–1080, 1988.
8. Yoshiyuki Koseki. Experience Learning in Model-Based Diagnostic Systems. *Proc. IJCAI*, pages 1356–1362, 1989.
9. Lukas Kuhn and Johan de Kleer. Diagnosis with Incomplete Models: Diagnosing Hidden Interaction Faults. *Proceedings of the 21st International Workshop on Principles of Diagnosis*, pages 1–8, 2010.
10. Vladimir Lifschitz. What Is Answer Set Programming? *23rd AAAI Conf. on Artificial Intelligence2*, pages 1594–1597, 2008.
11. P. J F Lucas. Bayesian model-based diagnosis. *International Journal of Approximate Reasoning*, 27(2):99–119, 2001.
12. Oliver Ray and Antonis Kakas. ProLogICA: a practical system for Abductive Logic Programming. *Workshop on Non-Monotonic Reasoning*, 2006.
13. R Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.
14. R Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

Appendix: Springer-Author Discount

LNCS authors are entitled to a 33.3% discount off all Springer publications. Before placing an order, the author should send an email, giving full details of his or her Springer publication, to `orders-HD-individuals@springer.com` to obtain a so-called token. This token is a number, which must be entered when placing an order via the Internet, in order to obtain the discount.

8 Checklist of Items to be Sent to Volume Editors

Here is a checklist of everything the volume editor requires from you:

- ☐ The final LaTeX source files
- ☐ A final PDF file
- ☐ A copyright form, signed by one author on behalf of all of the authors of the paper.
- ☐ A readme giving the name and email address of the corresponding author.