Appeared in: 4th International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC '18)

Maintaining configuration knowledge bases: Classification and detection of faults

Christina Aigner and Florian Grigoleit

Fakultät für Informatik, Technische Universität München,
Boltzmannstraße 3, 85748 Garching bei München
`{ch.aigner,florian.grigoleit}@tum.de`

Abstract. Collaborative embedded systems, for example adaptive factories, frequently adapt to or are adapted for new tasks or contexts. The variability of such systems, that is the space in which the system can adapt, is represented in variability models. Variants, concrete realizations from the variability space, are created either manually or automatically by configuration systems. In both cases, the correctness and quality of the variants relies on well-designed and well-maintained variability models or configuration knowledge bases. Designing and maintaining knowledge-bases are error-prone activities. Once faulty entries exist, they are difficult to detect and can result in anomalous system output, for example in the generation of faulty or not-optimal results. For configuration systems, this means incorrect or non-optimal configurations or no configurations at all for inconsistent knowledge bases. We work on techniques for automatically analyzing configuration knowledge bases on their correctness and consistency. In this paper, we present a taxonomy of anomalies in knowledge-bases of our configuration system -GECKO. The taxonomy describes classes of anomalous behavior and its causes. Finally, we sketch heuristics to detect such behavior and localize faulty entries in a knowledge base.

Keywords: Collaborative Embedded Systems, Variability Models, System configuration, fault analysis

1 Introduction

Dynamic systems change in structure, constituents, and functionality. The space in which they can change is called variability. Variability is the system quality that describes which parts are invariant and which are variable, within which limits and which components are available, [3]. Variability models represent functionality and variability of a system. One common approach for variability modeling are feature models. Knowledge-based configuration uses such models to generate configurations of variable systems [7]. Generating configurations requires that these models are well-designed and correctly represent in the modeled system family and its application domain. In complex variable systems where constituents and functions change frequently, such as cyber-physical-systems or collaborative embedded systems, it is even more important to ensure that knowledge-bases used for system configuration are correct and allow generating consistent configurations. If this wasn't the case, reconfiguration or adaptation would require additional time to check the consistency of newly generated configurations.

In a knowledge-based system, the formal representation of a problem and domain knowledge is called the knowledge base. If a knowledge base contains errors, is inconsistent, or is incomplete, a problem solver either produces an incorrect or non-optimal result, or no result at all. Incomplete or incorrect entries in the knowledge base emerge during its creation, extension, or maintenance. Especially in commercial tools, like configuration systems, containing hundreds or thousands of components, user requirements, and restrictions, the content of a knowledge base is frequently modified. To reduce the number of errors, and thereby the amount of effort to maintain the correctness and the consistency of the knowledge base, we are developing an inspection tool for detecting and identifying knowledge base errors in configuration system, specifically for GECKO [12].

To provide a fault detection mechanism for constraint-based configuration knowledge bases, we categorize faults and define classes to describe error states and their cause-effect behavior. So far, there has been little research on detecting knowledge base faults together with resolving detected faults in an efficient manner, for example by proposing resolving options to the user. To bridge the gap between detecting and resolving faults, we use consistency checks [9] to determine whether the configuration problem is still solvable after an engineer changes the knowledge base. Generally, the solvability of a configuration problem depends on whether all the active goals (or requirements) and sub goals of the problem can be satisfied. This satisfaction hierarchy constitutes the core of our inspection tool.

While creating or maintaining a knowledge base, the engineer adds components, goals and constraints, changes attributes or deletes constraints. During this process, errors emerge easily and resulting faults can remain hidden until they cause an incorrect or sub-optimal result. We develop a set of techniques which check the knowledge base, examining which entries of the knowledge base may have caused the fault. The techniques also should provide (partially) automated support to localize the fault in case of non-solvability of the configuration problem. This inspection can be made after the initial creation of the knowledge base or later in the life-cycle when the knowledge base is updated to meet new or altered application requirements. We emphasize internal inconsistency or incorrectness which lead to non-achievability of goals and therefore, to no solution of the constraint satisfaction problem.

Next, in section two, we present the configuration system GECKO and related work on detecting errors in knowledge bases. In section three, a taxonomy of knowledge base errors will be presented which laid the foundation for developing our inspection tool. This is followed by an example from [1] to illustrate our work. Section 5 presents an approach to detect and locate inconsistencies in a GECKO knowledge base. We conclude with a summary of our work and an outlook on further research.

2 Related work

2.1 GECKO: A generic, constraint-based configuration system

GECKO provides global optimization of configurations by solving configuration problems with optimal constraint satisfaction, [12]. GECKO's core is a variant of conflict-directed A* (CDA*) search. CDA* improves constraint-based best-first state-search by removing super-sets of inconsistent states and thereby pruning the search space, [14]. In the following, we provide a brief overview on GECKO concepts necessary for detecting anomalies in GECKO knowledge-bases.

Optimal Constraint Satisfaction Problem: A finite CSP is a triple *(V, D,C)* with a set of variables V, domains D (a domain is a finite set of values), and constraints C where each domain dom(vi) represents all valid assignments for a variable vi, [4]. The set C contains all constraints which restrict the number of valid solutions of a configuration problem. A constraint defines a set of valid value assignments a for variables and relations between them. The set A(ci) is the set of assignments a constraint ci has. An assignment a to variables var(cj) satisfies the constraint if a $\in$ cj, and violates it otherwise. An optimal constraint satisfaction problem is a CSP in which solutions have different costs or utilities depending on the values of a (sub)set of variables, called decision variables. Thus, an optimal CSP consists of a triple *V,D,C* and a utility or cost function h(d), where d is a set of decision variables. The objective in optimal constraint satisfaction is maximizing (or minimizing) the utility (or cost) of the solution.

Knowledge-based Configuration: As defined in [13]: configuration is the activity of assembling a customized system from standard components. The task of a knowledge-based configuration system is to generate configurations according to user requests. In GECKO, we define a configuration problem as a triple (ConfigKB, Task, AA), where ConfigKB is a set of constraints representing the domain knowledge. Task is a set of clauses representing the requirements and restrictions on the configuration, and AA is a set of activity assignments on a catalog of components.

GECKO Concepts

GECKO Goal: Goals express the achievements expected from a configuration. They may have an associated priority, describing the importance of a configuration. Goals, if achieved, can contribute to other goals, allowing a hierarchy of goals, and the representation of goal refinement. A goal is achieved by another goal, or a component, or the combined contribution of multiple goals or components.

GECKO Components: Components represent the building blocks of the configuration. A component can either (partially) achieve goals or contribute to the usage of another component, a battery, for example, could enable the use of a flashlight. A component consists of a set of component attributes, specifying its characteristics, a component type, and set of contributions to goals or to other components.

GECKO Choice: 1:1 relations between goals and components, e.g. Goal1 requires Component1, or goals and other goals are rare. In variable systems, goals can be achieved by different components or sets of components. To represent variability, we introduced the concept Choice, i.e., n:m relations between goals/components and other goals/components. A choice consists of a set of options, a combine function, and a satisfaction threshold. The combine function combines the contributions of the choices active elements and is defined domain specific. In case a single option has to be selected, the combine function would be equivalent to an XOR gate, while in case a specific contribution level has to be achieved, the combine function could be a sum function of the contributions of the achieved / active options.

GECKO Task: A configuration task specifies a single configuration problem. A task is a triple (TaskGoals, TaskParameters, TaskRestrictions). TaskGoals are user selected goals a solution has to satisfy. TaskParameters are value assignments to parameters from the domain, for example the size of the factory under design. TaskRestrictions are explicit restrictions on the solution, such as the available resources or type of machine that is expected. A configuration is a solution to a Task, i.e., it achieves all Goals and is consistent with the domain knowledge represented in the configuration knowledge base ConfigKB.

2.2 Related work in fault detection:

The importance of correct knowledge bases made automatic fault detection tools a major topic in the development of knowledge based systems, particularly for configuration systems. Consistency-based diagnosis has been shown to be a promising approach for detecting faults in knowledge bases and explaining the unexpected behavior of the configuration system. Felfernig et. al. use precise behavior descriptions of the configuration to explain inconsistencies in the knowledge base. Positive and negative test examples identify possible faulty components and clauses [5]. Positive configuration examples should be accepted by the configuration system whereas negative examples should be rejected. The approach eventually aims at identifying a diagnosis which is a subset of the knowledge base. To repair the knowledge base, the diagnosis consists of an extension, which, when added to the knowledge base, restores the inconsistency for all negative examples.
Without additional domain knowledge, programs usually fail to detect semantic incorrectness, as they lack human reasoning abilities. Test cases are well-suited to identify incorrectness faults, due to the entailed domain knowledge in the test

examples. However, it requires significant effort to generate and execute test cases.
Other approaches for fault detection search for inconsistencies arising through added or changed constraints in the knowledge base. Algorithms like FAST-DIAG [8] produce a minimal diagnosis of inconsistent constraint sets. Other approaches have been presented to support the engineer in his task of knowledge base maintenance. For example techniques for detecting well-formedness violations [16], simulation [15], or knowledge base verification. These algorithms reduce the time to detect the anomalies and explain the anomaly to get a higher understanding of the knowledge base.

3 Taxonomy of faults

To develop a tool for detecting faults, we first analyzed the fault structure of GECKO to understand how faults emerge and how they effect the configuration. We found that the best solution technique for a fault varies significantly between the different fault categories. So we further investigated on the cause and effect relationships to distinguish between the error categories and faulty states of the configuration. Furthermore, we conducted a literature review on diagnosis and inspection of knowledge bases to create a systematic taxonomy of faults. We found that, the most frequently mentioned fault types in knowledge bases fall into three categories: 1) Inconsistency errors [5] [14] 2) Incompleteness errors [11] and 3) Redundancy errors [9] [15]. Moreover, there are knowledge-based anomalies in the sense of a well-formedness violation of the knowledge base [16]. These anomalies can include dead or full mandatory domain elements or implicit constraints. For our work, we decided that these categorizations and definitions must to be unified in a single taxonomy to describe the correlation between faults and anomalies of a constraint-based system like GECKO. Figure 1 depicts the fault classes in relation with their cause and potential effects. We introduce three levels of knowledge-base faults: *Entries, Anomalies and Abnormal Output.*

Entry categories The first level (Entry) describes two cases how a person can enter content to or edit content in a knowledge base. This can either be a non-faulty entry (semantically and syntactically correct) or a faulty entry, thus, an error made by human mistake. We introduce the following subcategories of entries: An error can be an incorrect entry (semantically or syntactically incorrect), an incomplete entry (an unfinished entry or entry where parts are missing) or an inconsistent entry (stands in conflict with the existing and internally consistent knowledge base). If the entry is correct, we still have to distinguish between a redundant or non-redundant entry; in other words, whether the entry changes the knowledge representation or is superfluous [9].

Anomalies On the second level, we examined the effect of errors and the structure of the abnormal or problematic states an error can lead to. We refer to

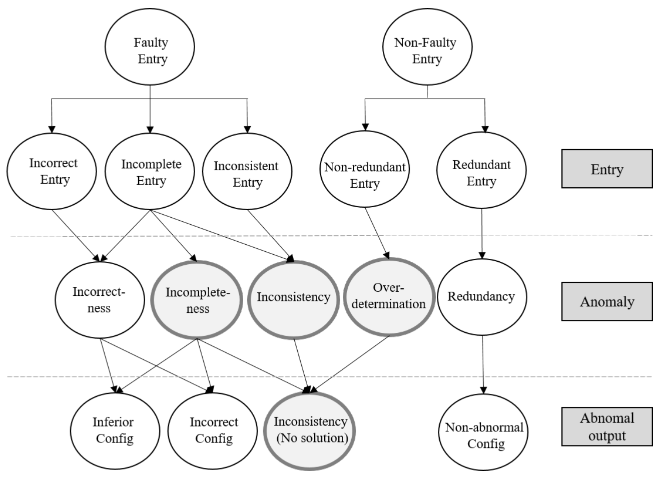


Fig. 1: Fault taxonomy

these states as anomalies. Anomalies are caused by anomalous entries (second level in Fig 1 and can result in several types of abnormal outputs. We identified five anomalies: inconsistency, incorrectness, incompleteness, over-determination, and redundancy. Inconsistencies are elements in the knowledge base which result in contradictory inferences and therefore prohibit valid solutions. Incorrectness describes the state when there is a domain element with a semantically wrong value which does not describe the domain knowledge correctly. The element is therefore inconsistent with the represented domain, i.e. a configuration based on incorrect entries is consistent with the knowledge base (solvable), but semantically incorrect. A knowledge base can be semantically incorrect if i) there exists a value of a variable domain which fails to represent the domain knowledge correctly or, ii) there exists a constraint which represents an incorrect relation between structures in the knowledge base. In both cases, the incorrect entries result in a configuration that is either non-optimal or semantically inconsistent with the application domain. Incompleteness refers to the extension, degree, amount or coverage to which the information in a user-independent ontology covers the information of the real-world domain [10]. A knowledge base is in an incomplete state if there exists another domain element, which is necessary to restore alignment with the real-world domain. Incompleteness is an anomaly which does not necessarily lead to an inconsistency itself but can result in multiple abnormal output states, namely inconsistency, incorrectness, or inferiority. A knowledge base contains redundancy if there is a constraint c which can be removed from the knowledge base without changing the semantics of the configuration. By deleting redundant constraints from a knowledge base, only a set of non-redundant constraints remains, the minimal core. There, the knowledge base contains an element which is unnecessary to express the complete domain knowledge [7] [9]. Over-determined CSPs are problems for which no solution exists without the relaxation of one or more constraints [2]. Over-determined problems are not an anomaly per se, but require techniques for constraint relaxation, [4].

Abnormal outputs Anomalies can result in a fault in the fourth level of the taxonomy in Fig. 1: to an abnormal output. Abnormal outputs of a configuration system mean that the CSP either computes an incorrect solution, an inferior solution or no-solution at all. A CSP has no-solution to a configuration task if there is no available choice of components which can satisfy the task goals of the user. A configuration is an incorrect solution to a CSP if it contains components which are not part of a semantically sound solution.

With this taxonomy, we reanalyzed GECKO knowledge bases to find a more specific description of the behavior of faults in the systems and to find patterns of faults emergence. We analyzed on the following levels of inspection: (1) the source of the fault and the kind of entry related to the fault, (2) the anomaly the entry leads to, (3) the dependencies or other factors the anomaly is influenced by (4) whether the anomaly computes an abnormal output. We found that in GECKO, incorrectness and inconsistency faults can source from various areas in the knowledge base. However, these faults were mainly caused by faulty entries in the process of choice achievement.

4 Example

One application of GECKO is the adaptable and flexible factory, [1]. Here, we describe an example from such a factory. In an adaptable and flexible factory, variability is given, by both the option to easily change production lines by adding or removing manufacturing machines, and by changing the operating mode of machines so that they can provide more than one manufacturing function. For demonstration, we use a simple configuration problem from the area of manufacturing and production. A product can be manufactured in various variants. Depending on the purpose of the product and the future tasks the product should be able to perform, specific features have to be included or excluded. A viable, optimal combination is required for the product fulfill its purpose. This optimization can be applied on the hardware and on the software level. The features need to be compatible with each other and can require or exclude each other. This can be represented with domain constraints. Moreover, to perform a production task, a variety of smaller sub tasks have to be processed. We represent these sub tasks as sub goals (goals required by another goal) of the main user goal. All steps of the production process (sub goals) need to be achieved to fulfill the main user goal. In this example, we will insert a faulty constraint to the knowledge base to show how a resulting failure could be diagnosed.
The following working example shows a simplified configuration knowledge base of a manufacturing system. We explicitly omit software components and system collaboration to ensure the simplicity and understandability of the example. Our

manufacturing system consists of three subsystems: drilling, milling, and welding. Here, we look into the subsystem of drilling. The goal of the configuration is to assemble the production equipment to fulfill the task of the user to drill a hole. The users can specify preferences and restrictions of his task (e.g. size and depth of the whole) which are noted as parameters. These parameters restrict the solution space by adding parameter constraints the configuration task.
In our subsystem we have one selected main user goal: *Drill a hole.* Achieving the main user goal *drill a hole*, requires all its sub goals to be achieved. In this example, the sub goals are the *positioning of the object* and *assembly of the drilling machine* are required. For the assembly of the drilling machine, we further need to find a consistent combination of a drilling machine and a drilling head for our task. Figure 2 illustrates the goal hierarchy of our system. This hierarchy implies the following Goal Constraints, which describe the requires-relation between them.

$Positioning.achieved \rightarrow Fixation.achieved \rightarrow Allocation.achieved$
$Assembly.achieved \rightarrow SetDrillingMachine.achieved \wedge SetDrillingHead.achieved$
$HoleDrilled.achieved \rightarrow Assembly.achieved \wedge Positioning.achieved$

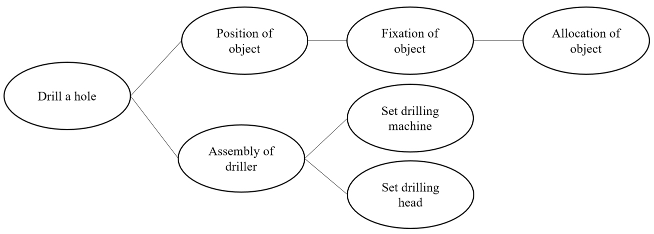


Fig. 2: Goal tree of user goal: drilling hole

There are two types of drilling machines modeled as a component: *box column drill* and *pillar drill.* Both contribute to the component choice *DrillingMachine-Choice.* This choice can be described as an XOR relationship, which means that only one active component can be chosen as a head to fulfill the goal *SetDrilling-Head.*

$BoxColumnDrill.contributesTo(DrillingMachineChoice) \wedge$
$PillarDrill.contributesTo(DrillingMachineChoice)$
$\rightarrow DrillingMachineChoice = BoxColumnDrill \wedge PillarDrill$

The drilling machines all need a drilling head to perform the task of drilling. Drilling heads can be chosen from a *DrillingHeadChoice*, which contains various different drilling heads with different attributes.

DrillingHeadChoice = Head1 $\vee Head2 \vee Head3 \vee ... \vee$ *HeadN*
Where for each Head i: Head.contributesTo(DrillingHeadChoice)

The machine-head relationship is depicted by a component constraint. Every drilling machine needs a drilling head. If there is no such drilling head active in the current configuration, the goal cannot be achieved.

PillarDrill.requires(DrillingHeadChoice)

Furthermore, there is a Gripper which should fix the product which needs to be drilled.

GripperChoice = Electrical $\vee$ *Pneumatic*

In addition to the component choices, components can have attributes. Our drilling heads have the attributes length, diameter and headtype. Their values are specified in the knowledge base. For simplicity, we only specify one drilling head in greater detail:

AttributeVariables = diameter, length, headtype
Dom(diameter) = 3, 6, 8
Dom(length) = 40, 60, 80, 100
Dom(headtype) = slot, hex-socket, frearson
Head1.diameter = 5mm
Head1.length = 100mm
Head1.headtype = hex-socket

Furthermore, we can specify invalid combinations of attributes which are described by Component Constraints: Drilling heads of 6mm diameter cannot have a length of 100mm, the system has no slot heads with a length of 40mm and frearson heads are always of 8mm diameter.

diameter = 3 $\rightarrow$ *b*$\neg$*blength = 100*
headtype = slot $\rightarrow \neg$*length = 40*
headtype = frearson $\rightarrow$ *diameter = 8*

If the user wants to compute a configuration, he must select a TaskGoal and TaskParameters. The user wants to drill whole into a metal plate and specifies the given task parameters:

Product.material = metal
holeDiameter = 6
holeDepth = 40

This specification of the task comes along with the following TaskParameterConstraints:

$drillhead.diameter = holeDiameter$
$drillhead.length = holeDepth$

During knowledge-base maintenance, additional constraints are added. Thereby, we demonstrate different types of faulty entries which could be added to the system. The engineer adds another drilling head - component Head6. A new component entry in our case requires three sub entries: attributes length, diameter, and headtype. We specify:

$Head6.headtype = slot$
$Head6.diameter = 8$
$Head6.contributesTo(DrillingHeadChoice)$

However, no length was specified. Hence, Head6 is an incomplete entry and therefore, cannot be used. Head6 will never be part of a consistent configuration.

A new component Hammer is added to the knowledge base.

$Hammer.contributesTo(DrillingMachineChoice)$

This represents an incorrect entry, because a hammer is not a drilling machine. If the component *Hammer* is active, it can satisfy the *DrillingMachineChoice* and thus, can be computed as an output for our drilling task. Obviously, drilling a hole with a hammer is an incorrect solution and the configuration output will be incorrect. As we can see here, the incorrect entry does not necessarily lead to a non-solvable problem, but can compute a incorrect configuration.
An additional component constraint was added:

$headtype{=}slot \rightarrow length = 40$

This entry is inconsistent with our existing constraint if headtype = slot then $\neg$length = 40 and leads to a conflict if we select headtype = slot.
No configuration can be computed as length = 40 and length = $\neg$40 cannot be fulfilled at the same time.

5 Solution approach

Here, we describe an approach to detect inconsistency faults in the knowledge base. The presented algorithm can detect non-achievable choices in a GECKO knowledge-base. The defective choices need be revised to restore the consistency of the system.

5.1 Achievability checks

This approach focuses on the detection of faults leading to an inconsistent configuration, that is a state of the system where no solution can be generated for a configuration task (see figure 1 - abnormal outputs). The other two output categories, inferior and incorrect configurations, are not covered in this approach. To find the responsible entry for an inconsistency, we have to take a closer look at the anomalies which lead to an inconsistent configuration. Our taxonomy proposes that the following anomalies can result in a non-solvable configuration problem: **(1) over-determination (2) incompleteness and (3) inconsistency** (see figure 1 - marked). Hence, these are the anomalies which have to inspected.
(1) As over-determination is not directly a faulty behavior, but rather the state of too many restrictions in the system, it can also be a desired behavior. Therefore, we decided to focus only on the main core of the knowledge base including goal and component constraints of GECKO and to rule out additional task parameter constraints, as they only increase the amount of conflicts which have to be inspected without providing additional information on the true source of an inconsistency. Task Parameter constraints should be added after a first internal check of the knowledge base, in the form of test cases.
(2) Incompleteness regarding the choice options (to make a choice generally be achievable) can easily be checked by adding up all contributions of the potential options without taking any further constraints into account. If there are not enough options for a choice to ever surpass its achievability threshold, then this choice is marked as incomplete.
(3) To inspect the knowledge base for cases of inconsistency, we want to diagnose faults in the configuration knowledge base while computing a configuration for a specific user goal. In contrast to existing approaches for fault localization and diagnosis, we use hierarchy achievability checks, to determine if the choices of the goals and its sub goals are achievable with the available components of the knowledge base and under the given constraints. Through exploiting the hierarchical structure of goal satisfaction in GECKO we can divide the computation problem of a diagnosis into smaller sub problems. Our algorithm uses a systematic hierarchical approach by checking the generated choice tree. Using the hierarchical goal structure for diagnosis is a known concept that has shown advantages in coping with the complexity of large configuration knowledge bases [6].
Hierarchies have also been used to compute minimum cardinality diagnosis like in [17], who achieved significantly higher efficiency and scalability as opposed to a traditional diagnosis.
KBIn represents a quick and effective diagnosis tool which is added to the GECKO system while developing the knowledge base. KBIn verifies the achievability of a certain configuration task at three different points of change: at initial knowledge base creation, knowledge base extension or maintenance. When the knowledge base is updated, KBIn makes sure the changes do not impair the achievability of the system and checks whether the knowledge base is still consistent after the changes. As we want to keep KBIn a quickly executable tool,

the user can choose the top-level goal which needs to be inspected. In this way, it's not necessary to deploy the whole knowledge base, but rather, a relevant sub set of it. Therefore, KBIn makes it possible to focus on those elements in the knowledge base which are related to the change. The achievability check uses a top-down approach, starting with the selected high-level user goal and refining the results down to the node where the change takes effect. This guarantees the usability of the system while leveraging the time savings through skipping all unrelated parts in the knowledge base.

5.2 Algorithm

The algorithm starts to inspect the choice tree if no solution can be found for a the selected user goal. In case of non-achievability of the first choice, KBIn takes a closer look into the related options (sub choices) (1).

(1) If Choice.achieved $= F \wedge \exists$ *option then*
$\forall$ *options of choice: check achievability* (see figure 3)

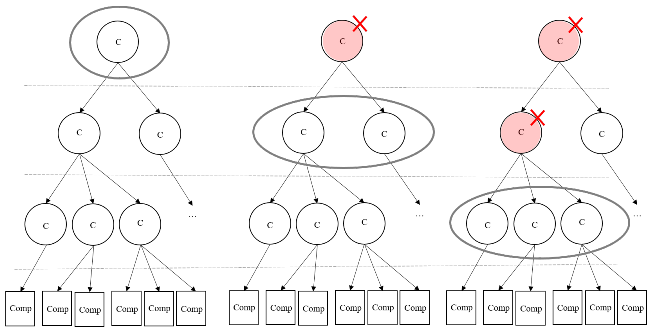


Fig. 3: Recursion levels - step in when not achievable

Furthermore, we distinguish two cases of non-achievability, depending on whether the inconsistent choice is a parent node (with children) or a leaf node (end node without children) of the choice tree. Non-achieved leaf node choices are always added to the set of inconsistent choice as we know that the inconsistency lies within this choice, because there are no more sub choices which could be an explanation for the inconsistency. (2) If a parent node choice cannot be achieved, all its options have to be inspected (see definition (1) and method checkChildren in figure 5). In case all these options are achieved, the inconsistency cannot be explained by its options, so that the choice has to be added to the set of inconsistent choices (3) (in figure 5 the choice is appended to the list of inconsistent choices).

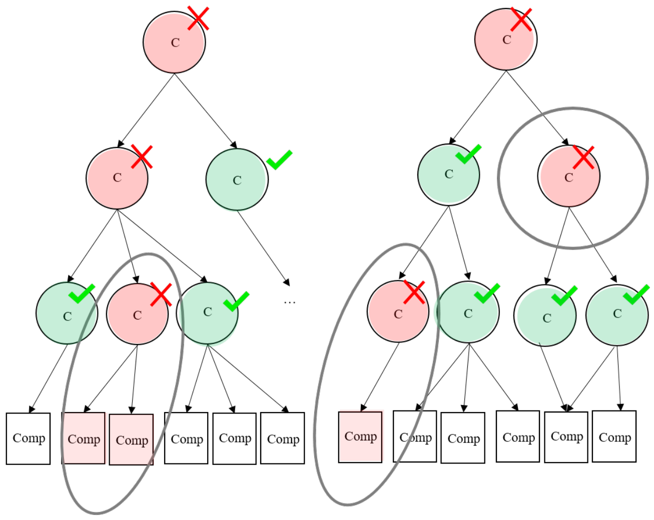


Fig. 4: Return inconsistent end node choice (left), return inconsistent parent node choice (right)

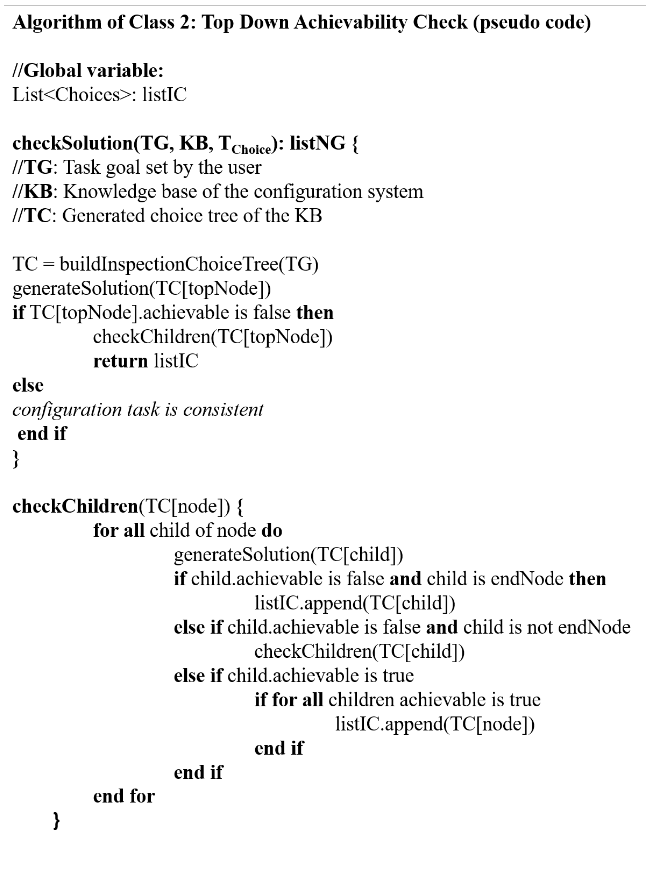
(2) If Choice.achieved $= F \ \wedge \neg\exists$ option then
choice $\vDash \bot$

(3) If Choice.achieved $= F \ \wedge \forall$ options: option.achieved = T then
choice $\vDash \ \bot$ (see figure 4)

The algorithm in figure 5 implements the above concepts and functionality (visualized in figure 4 and 3) in pseudo code. The following variables and concepts are used: A List *listIC* collects the set of inconsistent choices which cannot be achieved under the current task goal. After the achievability check this list will be returned to the user or passed on to further diagnosis. The function *buildChoiceTree* generates a choice tree of the all GECKO choices involved in achieving the top-level task goal selected by the user. The function *generateSolution* solves the configuration problem for the top-level task goal (= topNode) and with this, computes all goals and choices of the choice tree. The choices can either be achieved (node.achievable = T) or not achieved (node.achievable = F). The concept *endNode* describes a node of a choice tree, which is a child without any further sub options (or sub goals), e.g. a component choice. In other words a leaf node of a choice tree.

5.3 KBIn achievability check with Example

First, a configuration for the user goal *Drill a hole* is computed. Since the configuration systems fails to produce a solution, we refine the search and check all underlying options in the choice tree, which are: *Positioning* and *Assembly.* The



```
Algorithm of Class 2: Top Down Achievability Check (pseudo code)

//Global variable:
List<Choices>: listIC

checkSolution(TG, KB, $T_{Choice}$): listNG {
//TG: Task goal set by the user
//KB: Knowledge base of the configuration system
//TC: Generated choice tree of the KB

TC = buildInspectionChoiceTree(TG)
generateSolution(TC[topNode])
if TC[topNode].achievable is false then
        checkChildren(TC[topNode])
        return listIC
else
configuration task is consistent
 end if
}

checkChildren(TC[node]) {
        for all child of node do
                generateSolution(TC[child])
                if child.achievable is false and child is endNode then
                        listIC.append(TC[child])
                else if child.achievable is false and child is not endNode
                        checkChildren(TC[child])
                else if child.achievable is true
                        if for all children achievable is true
                                listIC.append(TC[node])
                        end if
                end if
        end for
    }
```

Fig. 5: Pseudo code algorithm of KBIn achievability check

algorithm proceeds with the choice *Positioning* and confirms its achievability. Therefore, the algorithm skips the underlying goals and continues with the next child on the same level, *Assembly*. *Assembly* cannot be achieved, so we compute all options, which are: *Set drilling machine* and *Set drilling head*. As *Set drilling machine* is achieved, *Set drilling head* will be computed next and results in an inconsistency. As *Set drilling head* has no further children, hence, it is added to the set of inconsistent nodes and will be returned to the user.

The returned set contains all relevant constraints which have an impact on whether the choice can be achieved. The following list is returned to the user:

Set drilling head
$\mathit{diameter} = 3 \rightarrow \neg \mathit{length} = 100$
$\mathit{headtype} = \mathit{slot} \rightarrow \neg \mathit{length} = 40$
$\mathit{headtype} = \mathit{frearson} \rightarrow \mathit{diameter} = 8$
$\mathit{headtype} = \mathit{slot} \rightarrow \mathit{length} = 40$

Here, the two slot constraints are inconsistent and are the source of the inconsistency of the main user goal and further, the configuration.

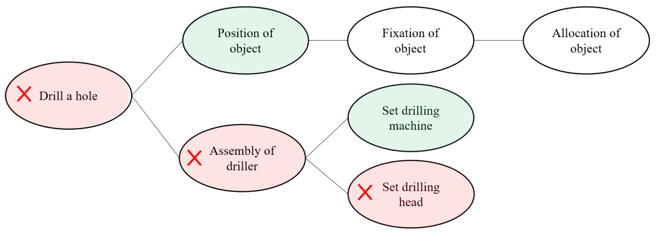


Fig. 6: Goal tree after achievability check

6 Conclusion

In this paper, we gave a summary of the state-of-the-art in techniques for describing, detecting, and identifying inconsistency and incorrectness in knowledge bases and provided an introduction to our work on automatic anomaly detection and identification for GECKO. We created the taxonomy for anomalies and anomalous results. Furthermore, we evaluated approaches for detecting different anomaly types, like implicit constraints or redundancy. Based on the taxonomy and our study of existing techniques, we outlined a set of techniques for detecting and identifying further anomalies in GECKO knowledge bases. Next, we will evaluate these techniques on at least two GECKO knowledge-bases from different application domains and develop a more detailed diagnosis algorithm, which does not only detect the inconsistent choices but also explains which constraint or entry is responsible.

Acknowledgments. We would like to thank Wolfgang Böhm and Peter Struss for their valuable feedback. This research has in part been funded by the *German Federal Ministry of Education and Research* under the grants no. 01IS16043, 01IS16043, and 01IS16043.

References

1. Collaborative Embedded Systems (CrESt). `https://crest.in.tum.de/`, 2017. [Online; accessed 01-July-2017].
2. R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. *IJCAI*, 93:276–281, 1993.
3. D. Beuche and M. Dalgarno. Software product line engineering with feature models. In *Overloard Journal*, pages 5–8, 2007.
4. R. Dechter. *Constraint processing.* Morgan Kaufmann, 2003.
5. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213–234, 2004.
6. A. Felfernig, G. Friedrich, Jannach D. Zanker, M., and M. Stumptner. Hierarchical diagnosis of large configurator knowledge bases. *Advances in Artificial Intelligence*, pages 185–197, 2001.
7. A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 26(1):53–62, 2012.
8. A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM 26*, 1:53–62, 2012.
9. A. Felfernig, C. Zehentner, and P. Blazek. Corediag: Eliminating redundancy in constraint sets. *DX International Workshop on Principles of Diagnosis*, 22:219–224, 2011.
10. A. Gomez Perez. Some ideas and examples to evaluate ontologies. In *In Artificial Intelligence for Applications, 1995. Proceedings., 11th Conference on*, pages 5–10. IEEE, 1995.
11. A. Gomez Perez. Evaluation of ontologies. *International Journal of intelligent systems*, 16(3):391–409, 2001.
12. F. Grigoleit and P. Struss. Configuration as diagnosis: Generating configurations with conflict-directed a* - an application to training plan generation. In *DX@ Safeprocess, International Workshop on Principles of Diagnosis*, pages 91–98. DX, 2015.
13. U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. 2004.
14. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
15. F. Reinfrank, G. Ninaus, F. Wotawa, and A. Felfernig. Maintaining constraint-based configuration systems: Challenges ahead. In *Configuration (Workshop)*, 2015.
16. F. Reinfrank, G. Ninaus, F. Wotawa, and A. Felfernig. Intelligent supporting techniques for the maintenance of constraint-based configuration systems. In *Configuration (Workshops)*, pages 31–38, 2015.
17. S. A. Siddiqi and J. Huang. Hierarchical diagnosis of multiple faults. In *International Joint Conference on Artificial Intelligence*, volume 16(3), pages 581–586. IJCAI, 2007.