**DIRECTDEBUG: Automated Testing and Debugging of Feature Models**

Viet-Man Le  
*Institute of Software Technology*  
Graz University of Technology  
Graz, Austria  
vietman.le@ist.tugraz.at

Alexander Felfernig  
*Institute of Software Technology*  
Graz University of Technology  
Graz, Austria  
alexander.felfernig@tugraz.at

Mathias Uta  
*Siemens Energy AG*  
Erlangen, Germany  
mathias.uta@siemens.com

David Benavides  
*Computer Languages and Systems*  
University of Seville  
Seville, Spain  
benavides@us.es

Jose Galindo  
*Computer Languages and Systems*  
University of Seville  
Seville, Spain  
jagalindo@us.es

Thi Ngoc Trang Tran  
*Institute of Software Technology*  
Graz University of Technology  
Graz, Austria  
ttrang@ist.tugraz.at

Abstract—Variability models (e.g., feature models) are a common way for the representation of variabilities and commonalities of software artifacts. Such models can be translated to a logical representation and thus allow different operations for quality assurance and other types of model property analysis. Specifically, complex and often large-scale feature models can become faulty, i.e., do not represent the expected variability properties of the underlying software artifact. In this paper, we introduce DIRECTDEBUG which is a direct diagnosis approach to the automated testing and debugging of variability models. The algorithm helps software engineers by supporting an automated identification of faulty constraints responsible for an unintended behavior of a variability model. This approach can significantly decrease development and maintenance efforts for such models.

Index Terms—Automated Testing and Debugging, Feature Models, Variability Models, Diagnosis, Conflicts, Configuration.

I. INTRODUCTION

Feature models support the representation of variability and commonality properties of software artifacts [1], [2]. Applications thereof support users in deciding about which features should be included in a specific software instance. These models can be differentiated with regard to the used knowledge representation. So-called basic feature models [2] support the representation of hierarchies including cross-tree constraints such as excludes and requires relationships. Cardinality-based feature models [3] extend basic ones with cardinalities (> 1) of feature relationships. Finally, extended feature models [4] support the description of features with attributes.

The creation and evolution of feature models can be error-prone where cognitive overloads or missing domain knowledge are major reasons for models that do not reflect the intended variability properties [5]–[7]. Consequently, feature model development has to be pro-actively supported by intelligent debugging mechanisms that support the automated detection of faulty constraints responsible for the unexpected behavior of a feature model knowledge base.

The remainder of this paper is organized as follows. After a discussion of related work (Section II), we introduce an example of a feature model (Section III). In this context, we provide a formalization of feature models as constraint satisfaction problems (CSPs). This formalization is used as a basis for a discussion of concepts supporting the automated testing (Section IV) and debugging (Section V) of feature models. In Section VI, we report initial results of a performance analysis of our approach. The paper is concluded with Section VII.

II. RELATED WORK

The state-of-the-art in feature model analysis and related tasks can be summarized as follows.

**Feature Model Analysis Operations.** Analysis operations help to assure well-formedness properties in feature models. For example, it should be possible that each feature of a model can be included in at least one configuration, i.e., there should not exist a feature which is inactive in every possible configuration. For a detailed discussion of analysis operations for feature models we refer to Benavides et al. [1].

**Conflict Detection.** A conflict set (conflict) can be regarded as a *subset that induces an inconsistency*. For example, Zeller et al. [6] propose the DELTA DEBUGGING algorithm which supports the determination of relevant subsets in test cases responsible for the faulty behavior of a software component. Following a similar objective, Junker [9] introduces the QUICKXPLAIN algorithm for the identification of subsets of constraints in a knowledge base responsible for an inconsistency (no solution can be found). Conflict sets are the basis for follow-up diagnosis operations that help to resolve these conflicts. More precisely, a diagnosis (hitting set [10], [11]) entails a set of elements of a knowledge base that have to be adapted or deleted to be able to resolve all conflicts, i.e., to restore consistency in the knowledge base. In contrast to conflict detection [8], [9], the approach presented in this paper focuses on diagnosis, i.e., conflict resolution.
Diagnosis of Inconsistent Models. A diagnosis can be regarded as a deletion subset that helps to restore consistency. An approach to the identification of diagnoses for inconsistent constraint sets is presented in Bakker et al. [12]. In this line of research, Trinidad et al. [6] show how to determine such diagnoses in the context of inconsistent feature models. In contrast to the work of Bakker et al. [12] and Trinidad et al. [6], the approach presented in this paper focuses on scenarios where test cases are used to induce an inconsistency in a knowledge base. Our approach is based on the idea of Felfernig et al. [13] who introduce a model-based diagnosis approach [11] to resolve conflicts in knowledge bases induced by test cases. Compared to Felfernig et al. [13], our approach is based on direct diagnosis (no conflict detection needed) which allows for an efficient determination of diagnoses [10].

Diagnosis for Reconfiguration. A reconfiguration can be regarded as a set of adaptations of feature settings in a changed configuration that are needed to restore consistency between the configuration and the corresponding feature model [14]. Using a constraint-based representation [15] of a feature model, White et al. [16] show how to apply the concepts of model-based diagnosis [11] to determine minimal sets of feature settings in existing configurations that need to be adapted in order to restore consistency with the feature model. In this context, feature models are assumed to be consistent. The work presented in this paper generalizes the concepts of Trinidad et al. [6] and White et al. [16] by allowing to take into account a set of test cases at the same time, i.e., a diagnosis represents an adaptation proposal that makes all of the given test cases consistent with the knowledge base.

Direct Diagnosis. The idea of direct diagnosis [10] is to significantly improve the performance of hitting set based diagnosis approaches [11] by supporting the calculation of diagnoses without the need of predetermining conflict sets. In this paper, we show how to support the automated testing and debugging of feature models using direct diagnosis [10]. The contributions of this paper are threefold. First, we show how to extend direct diagnosis [10] to support the automated testing and debugging of feature models. We integrate testing and diagnosis in a unified manner where test cases are considered as a central element of a diagnosis process. Second, we show how different types of test cases can be integrated into automated debugging processes. Third, we report initial results of a performance analysis of our diagnosis approach.

III. FEATURE MODEL SEMANTICS

Feature models are used to represent software variability properties by specifying features and their relationships in a hierarchical fashion [2]. Features are arranged in a hierarchical fashion with one specific root feature $f_r$ which has to be included in every configuration [1]. In a feature model, features are represented as nodes and relationships between features as corresponding edges. For an overview of approaches to the representation of feature models, we refer to Batory [4].

Feature Model Semantics. For the discussions in this paper, we follow the feature model representation of Benavides et al. [1] which includes four types of relationships (the hierarchical constraints mandatory, optional, alternative, or) and two types of cross tree constraints (requires and excludes). Feature models are variability models which can be formalized as a Constraint Satisfaction Problem (CSP) [13]. Each feature $f_i$ is related to the binary domain $\{ (true, false) \}$. The mentioned relationships and cross-tree constraints are represented as constraints on the CSP level.

Example Feature Model. Figure 1 depicts an example of a presumably faulty feature model (for details, see Section IV) from the domain of software services supporting the creation and management of surveys. For example, the feature ABtesting indicates whether a user wants to use AB testing functionalities when analyzing the results of a user study completed on the basis of questionnaires. Furthermore, the feature payment indicates the preferred payment mode where license represents a yearly payment and nolicense indicates a free license with an associated limited set of enabled features.

Constraint Types. The following semantics of feature model constraints is based on Benavides et al. [1].

Mandatory: feature $f_b$ is denoted as mandatory if it is in a mandatory relationship with another feature $f_a$. On the logical level, a mandatory relationship is defined in terms of an equivalence $f_a \leftrightarrow f_b$. In Figure 1 the feature Q&A is mandatory, i.e., it has to be part of every survey configuration. The same holds for payment and ABtesting where the latter should be considered faulty since, for example, it makes nolicense a dead feature and statistics a false optional [1].

Optional: if a feature $f_b$ is denoted as optional, this means that it may or may not be included in the case that feature $f_a$ is included. On the logical level, this property is formulated as implication: $f_b \rightarrow f_a$. In Figure 1 statistics is an optional feature connected to survey.

Alternative: exactly one feature $f_t$ out of $\{ f_1, \ldots, f_k \}$ has to be selected if feature $f_a$ has been selected. On the logical level, alternative relationships can be formalized as follows: $f_1 = t \leftrightarrow (f_2 = f \land \ldots \land f_k = f \land f_a = t) \lor \ldots \lor f_k = t \leftrightarrow (f_1 = f \land \ldots \land f_{k-1} = f \land f_a = t)$. An example thereof is payment with the subfeatures license and nolicense.

Or: at least one feature $f_t$ out of a feature set $\{ f_1, \ldots, f_k \}$ has to be included if feature $f_a$ has been selected. Relationships of type or can be formalized as follows: $f_a \leftrightarrow f_1 = t \lor f_2 = t \lor \ldots \lor f_k = t$. An example of a feature $f_a$ is Q&A, the subfeatures are multiplechoice and singlechoice.
Requiring: feature \( f_a \) must be included in a configuration if feature \( f_b \) is included. On the logical level, requires relationships can be defined as \( f_a \rightarrow f_b \). An example of a requires relationship is \( ABtesting \rightarrow statistics \).

Excludes: \( f_a \) and \( f_b \) must not be combined (\( f_a \) excludes feature \( f_b \) and vice versa). On the logical level, excludes relationships can be defined as \( \neg(f_a \land f_b) \). An example of an excludes relationship is: \( \neg(ABtesting \land \neglicense) \).

Feature Models and Configuration Tasks. The task of finding a solution for a constraint satisfaction problem representing a feature model can be interpreted as a configuration task (see Definition 1).

**Definition 1 (Configuration Task).** A configuration task \((F, D, C)\) is defined by a feature set \( F = \{f_1, f_2, ..., f_n\} \) and a set of feature domains \( D = \{\text{dom}(f_1), \text{dom}(f_2), ..., \text{dom}(f_n)\} \) \( \text{dom}(f_i) = \{(t)\text{rue}, (f)\text{alse}\}) \). Furthermore, \( C = CR \cup CF \) represents constraints restricting the possible solutions for a configuration task where \( CR = \{c_1, c_2, ..., c_k\} \) is a set of user requirements and \( CF = \{c_{k+1}, c_{k+2}, ..., c_m\} \) a set of feature model constraints.

In Definition 1, \( CR \) is an additional set of constraints which specifies which features should be included in a configuration. Based on Definition 1, we introduce the concept of a configuration (solution) for a configuration task (Definition 2).

**Definition 2 (Configuration).** A feature model configuration for a given feature model configuration task is an assignment \( A \) of all feature variables \( f_i \in F \). \( A \) is consistent if \( A \) does not violate any constraint in \( CR \cup CF \).

CSP Representation of a Feature Model. A CSP-based representation of a feature model configuration task \((F, D, C = CR \cup CF)\) that can be generated from the model shown in Figure 1 is the following. In this context, \( c_0 : survey = t \) is regarded as root constraint that is used to avoid the derivation of (irrelevant) empty configurations.

- \( F = \{\text{survey}, \text{payment}, \text{license}, \text{nolicense}, ABtesting, statistics, Q&A, \text{multiplechoice}, \text{singlechoice}\} \)
- \( D = \{\text{dom}(\text{survey}) = \{t, f\}, \text{dom}(\text{payment}) = \{t, f\}, ..., \text{dom}(\text{singlechoice}) = \{t, f\}\} \)
- \( CF = \{c_0 : survey = t, c_1 : survey \leftrightarrow payment, c_2 : survey \leftrightarrow ABtesting, c_3 : statistics \rightarrow survey, c_4 : survey \leftrightarrow Q&A, c_5 : Q&A \leftrightarrow \text{multiplechoice} \lor \text{singlechoice}, c_6 : (\text{license} \leftrightarrow \neg\text{nolicense} \land \text{payment}) \land (\text{nolicense} \leftrightarrow \neg\text{license} \land \text{payment}), c_7 : \neg(\text{ABtesting} \land \neg\text{license}), c_8 : \text{ABtesting} \rightarrow \text{statistics}\} \)
- \( CR = \{c_9 : \text{license} = t, c_{10} : \text{payment} = t\} \)

A consistent configuration that can be generated from our example feature model configuration task is the following:

- \( \{\text{survey} = t, \text{payment} = t, \text{license} = t, \text{nolicense} = f, \text{ABtesting} = t, \text{statistics} = t, \text{Q&A} = t, \text{multiplechoice} = t, \text{singlechoice} = t\} \)

Due to faulty constraints in a feature model, in some situations the constraint solver (configurator) does not determine the intended solution(s). We now show how to identify a minimal set of constraints in a feature model that need to be adapted (or deleted) to restore consistency with regard to a predefined set of test cases which specify the intended behavior of a feature model. In other words, we are interested in constraints responsible for the faulty behavior of a knowledge base generated from a feature model. In the following, we will show how to automatically determine such constraint sets on the basis of the concepts of direct diagnosis [10].

**IV. Testing Feature Models**

In Figure 1, ABtesting is mandatory, however, this triggers a situation where each configuration has to include statistics and it is not possible to generate a configuration with nolicense payment. Reasons for faulty feature models can range from misinterpretations in domain knowledge communication, modeling errors, to outdated parts of a knowledge base.

**Positive Test Cases.** The set of positive test cases \( T_p \) specifies the intended behavior of a knowledge base (feature model). Positive test cases \( t_i \in T_p \) are assumed to be existentially quantified, i.e., for each \( t_i \) there should exist at least one configuration consistent with \( t_i \). Such test cases can be derived from already existing consistent complete or partial configurations (e.g., represented by a set of included and excluded features), from a set of analysis operations (e.g., dead features), or specified by domain experts interested in the correctness of the feature model. Without loss of generality, we restrict our working example to positive test cases specifying the intended behavior of a knowledge base (see Table I).

| ID  | Test Case (Constraint)   |
|-----|-------------------------|
| 1   | nolicense = t           |
| 2   | license = t \& statistics = f |
| 3   | payment = f             |
| 4   | singlechoice = f        |

Avoiding nolicense to be a dead feature can be achieved by a positive test case \( t_1 : \text{nolicense} = t \) (nolicense should be included at least one configuration). Similarly, an example of a partial survey configuration could be: \( t_2 : \text{license} = t \& \text{statistics} = f \). Another test case could require the support of configurations with payment being deactivated: \( t_3 : \text{payment} = f \). Finally, we introduce a test case that assures the existence of a configuration where singlechoice is not included: \( t_4 : \text{singlechoice} = f \).

Note that test cases are not restricted to basic feature assignments but can also be implemented as general constraints, for example, we could specify \( ABtesting \land \text{license} \rightarrow \text{statistics} \) as a test case. In the following, we will integrate \( \{t_1...t_4\} \) in our discussion of a diagnosis algorithm that supports the automated testing and debugging of feature models.

**Negative Test Cases.** Negative test cases \( t_i \in T_{\neg} \) can be regarded as all-quantified constraints which specify an unintended behavior of a knowledge base. If \( t_i \) is unexpectedly consistent with the knowledge base, it is integrated in negated form into the background knowledge (see Section V).

**Generating Test Cases.** "Where do the test cases come from?" is an important question to be answered for applying the presented concepts in industrial settings. First, positive test cases can be derived from already completed and consistent feature model configurations. Second, positive as well as
negative test cases can be specified by domain experts. Third, negative test cases can be derived from inconsistent configurations, for example, configurations that have been identified as faulty by domain experts. Finally, test cases could also be directly generated from a feature model knowledge base, for example, by using well-formedness criteria from feature model analysis operations [1] (e.g., to avoid dead features \( f_i \), a test case \( f_i = t \) can be generated for each feature).

V. AUTOMATED DEBUGGING WITH DIRECTDEBUG

A diagnosis \( (\Delta) \) includes exactly those constraints responsible for the faulty behavior of a feature model. Intuitively, constraints in \( \Delta \) have to be deleted or adapted to make the feature model consistent with \( T_\pi \) (see Definition 3). Thus, a diagnosis helps a feature model engineer to focus diagnosis search. To enable such a functionality, test cases are needed.

**Definition 3 (Diagnosis and Maximal Satisfiable Subset).**

Given a feature model with a set \( CF \) of feature model constraints and a set of positive test cases \( T_\pi = \{ t_1, t_2, ..., t_l \} \). A diagnosis \( \Delta = \{ c_1, c_2, ..., c_k \} \) is a set of feature model constraints (\( \Delta \subseteq CF \)) such that \( \forall t_i \in T_\pi : \{ t_i \} \cup CF - \Delta \) is consistent. \( \Delta \) is minimal iff \( \neg \exists \Delta' \subseteq \Delta \) such that \( \forall t_i \in T_\pi : \{ t_i \} \cup CF - \Delta' \) consistent. A complement of \( \Delta \) (i.e., \( CF - \Delta \)) is denoted as Maximal Satisfiable Subset (MSS) \( \Gamma \).

**Example Diagnoses.** The minimal diagnoses that can be derived in our working example are depicted in Table II. There are two options for restoring the consistency between the feature model (Figure 1) and \{\( t_1, t_4 \): either delete/adapt the constraints in \( \Delta_1 \) or do this with the constraints of \( \Delta_2 \). For example, \( \Delta_1 \) suggests to take a look at \( c_1 \) (reconsider the relationship between \textit{payment} and \textit{survey}) and \( c_2 \) (reconsider the relationship between \textit{ABtesting} and \textit{survey}).

**Algorithm 1** DIRECTDEBUG(\( C = \{ c_1, c_2, ..., c_l \}, B, T_\pi : \Gamma \))

```
if isConsistent(\( C \cup B, T_\pi, T'_\pi \)) then
  return(\( C \))
end if
if \( |C| = 1 \) then
  return(\( \emptyset \))
end if
k ← \( \frac{l}{2} \)
\( C_1 \leftarrow c_1, c_2 \), \( C_2 \leftarrow c_k, ..., c_l \);
\( \Gamma_2 \leftarrow \) DIRECTDEBUG(\( C_1, B \cup T_\pi \));
\( \Gamma_1 \leftarrow \) DIRECTDEBUG(\( C_2, B \cup T_\pi \));
return(\( \Gamma_1 \cup \Gamma_2 \))
```

DIRECTDEBUG, all \( t_i \) in \( T_\pi = \{ t_1, t_q \} \) and \( T_\theta = \{ t_{q+1}, t_z \} \) have to be checked for consistency with \( C \cup B \).

If at least one positive test case induces an inconsistency in \( C \cup B \), DIRECTDEBUG is activated. Please note that only those test cases in \( T_\pi \) are forwarded to DIRECTDEBUG which are inconsistent with \( C \cup B \). In our setting, the original set of positive test cases \( T_\pi = \{ t_1, t_4 \} \) is reduced to \( \{ t_1, t_3 \} \) since \( t_3 \) does not induce an inconsistency in \( C \cup B \), i.e., there exists a configuration (solution) for \( C \cup B \cup \{ t_1 \} \).

DIRECTDEBUG determines a maximal satisfiable subset MSS (\( \Gamma \)) (Definition 3) where \( C \subseteq CF - \{ c_0 \} \) (consideration set), \( B = CF - C \cup \{ c_0 \} \cap T^- \) (background knowledge), and \( T^- \) represents a conjunction of negated negative test cases which are (unexpectedly) consistent with \( C \cup B \). DIRECTDEBUG is activated with test cases \( T_\pi \) that are inconsistent with \( C \cup B \). Given a set \( C \subseteq CF - \{ c_0 \} \) can be used to focus diagnosis search on specific parts of a feature model. If \( CF \) should be diagnosed as a whole, \( C = CF - \{ c_0 \} \) and \( B = \{ c_0 \} \cup T^- \).

**DIRECTDEBUG Consistency Checks.** IsConsistent checks whether the constraints in \( C \cup B \) are consistent with the test cases in \( T_\pi \). Obviously, test cases have to be checked individually, i.e., each activation of IsConsistent results in \( \{ T_\pi | \text{constraint solver activations (consistency checks)} \} \) returns \textit{t} (true) if every test case in \( T_\pi \) is consistent with \( C \cup B \), otherwise \textit{f} (false). Only test cases inducing an inconsistency with \( C \cup B \) are stored (returned) in \( T'_\pi \) (the remaining inconsistent positive test cases).

**DIRECTDEBUG Execution.** An execution trace of DIRECTDEBUG is shown in Figure 2. DIRECTDEBUG follows a divide & conquer approach. In each incarnation, it is analyzed which positive test cases remain inconsistent with \( C \cup B \). If just one constraint \( c_i \) remains in the consideration set \( C \) (\( |C| = 1 \)) and there still exists at least one test case \( t_j \) with \textit{inconsistent} (\( t_j \) \& \( C \cup B \)), then \( c_i \) is considered a part of a diagnosis \( \Delta \). If \( C \cup B \) is consistent with the test cases in \( T_\pi \), \( C \) is returned since no diagnosis elements can be found in \( C \).

**Diagnosis Determination.** DIRECTDEBUG returns a maximal satisfiable subset (\( \Gamma = \{ c_2, c_6 \} \)) (see Figure 2). To determine a minimal diagnosis \( \Delta \), we have to determine the MSS-complement, i.e., \( C - \Gamma \) which is \( \{ c_1, c_7, c_8 \} \).

VI. PERFORMANCE ANALYSIS

To evaluate the performance of DIRECTDEBUG, we synthesized test feature models (see Table III). For model synthesis,
we applied the BETTY generator [17] using the parameters #test positive cases ($\vert T_0\vert$), with a 30% share of inconsistency-inducing test cases) and #constraints in CF ($\vert CF\vert$), where #variables = $\vert CF\vert$ 2 with $C = CF - \{c_1\}$. Since each test case check needs a constraint solver call runtime increases with an increasing number of test cases and constraints. Each $\vert T_0\vert \times \vert CF\vert$ entry in Table III represents the average DIRECTDEBUG (diagnosis) computing time after 3 repetitions.

ACKNOWLEDGMENT

This work has been partially funded by the Horizon 2020 project OPENREQ (732463), the Austrian Research Promotion Agency PARXCEL project (880657), and the EU FEDER program MINECO project OPHelia (RTI2018-101204-B-C22).

REFERENCES

[1] D. Benavides, S. Segura, and A. Ruiz-Cortes, “Automated analysis of feature models 20 years later: A literature review,” Information Systems, vol. 35, pp. 615–636, 2010.
[2] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, “Feature-oriented Domain Analysis (FODA) – Feasibility Study,” TechnicalReport CMU – SEI-90-TR-21, 1990.
[3] K. Czarnecki, S. Helsen, and U. Eisenecker, “Formalizing Cardinality-based Feature Models and their Specialization,” SoftwareProcess: Improvement and Practice, vol. 10, no. 1, pp. 7–29, 2005.
[4] D. Batory, “Feature Models, Grammars, and Propositional Formulas,” in Software Product Lines Conference, ser. LNCS, H. Obbink and K. Pohl, Eds. Springer, 2005, vol. 3714, pp. 7–20.
[5] D. Benavides, A. Felfernig, J. Galindo, and F. Reinfrank, “Automated Analysis in Feature Modelling and Product Configuration,” in ICSR’13, ser. LNCS, no. 7925, Pisa, Italy, 2013, pp. 160–175.
[6] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro, “Automated error analysis for the agilization of feature modeling,” The Journal of Systems and Software, vol. 81, pp. 883–896, 2008.
[7] A. Zeller, “Automated debugging: are we close?” IEEE Computer, vol. 34, no. 11, pp. 26–31, 2001.
[8] A. Zeller and R. Hiieidebrandt, “Simplifying and isolating failure-inducing input,” IEEE Trans. on Software Eng., vol. 28, no. 2, pp. 183–200, 2002.
[9] R. Junker, “QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems,” in AAMAS 2004, 2004, pp. 167–172.
[10] A. Felfernig, M. Schubert, and C. Zehntner, “An efficient diagnosis algorithm for inconsistent constraint sets,” AI for Engineering Design, Analysis, and Manufacturing (AIEDAM), vol. 26, no. 1, pp. 53–62, 2012.
[11] R. Reiterer, “A theory of diagnosis from first principles,” Artificial Intelligence, vol. 32, no. 1, pp. 57–95, 1987.
[12] R. Bakker, F. Dikker, F. Tempelman, and P. Wogmim, “Diagnosing and solving over-determined constraint satisfaction problems,” in IJCAI’93, Morgan Kaufmann, 1993, pp. 276–281.
[13] A. Felfernig, G. Friedrich, D. Janneck, and M. Stumptner, “Consistency-based diagnosis of configuration knowledge bases,” Artificial Intelligence, vol. 152, no. 2, pp. 213 – 234, 2004.
[14] A. Felfernig, R. Walter, J. Galindo, D. Benavides, S. Erdeniz, M. Atas, and S. Reiterer, “Anytime Diagnosis for Reconfiguration,” Journal of Intelligent Information Systems, vol. 51, no. 1, pp. 161–182, 2018.
[15] E. Tsang, Foundations of Constraint Satisfaction. Acad. Press, 1993.
[16] J. White, D. Benavides, D. Schmidt, P. Trinidad, B. Dougerty, and A. Ruiz-Cortes, “Automated diagnosis of feature model configurations,” Journal of Systems and Software, vol. 83, no. 7, pp. 1094–1107, 2010.
[17] S. Segura, J. Galindo, D. Benavides, J. Parejo, and A. Ruiz-Cortés, “BETTY: Benchmarking and Testing on the Automated Analysis of Feature Models,” in Vamos’12. ACM, 2012, pp. 63–71.

VIII. DATA AVAILABILITY

The test dataset used for evaluation purposes can be found here: https://github.com/AIG-ist-tugraz/DirectDebug.

1For evaluation purposes, we used choco-solver.org.