



Viet-Man Le, MSc

Intelligent Techniques for Efficient Diagnostic Reasoning in Constraint-Based Systems

DOCTORAL THESIS

to achieve the university degree of
Doktor/Doktorin der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr. Alexander Felfernig
Institute of Software Engineering and Artificial Intelligence,
Graz University of Technology, 8010 Graz, Austria

Graz, May 2026

*To my parents,
who carried me through every page with quiet strength.*

*To Trang,
who turned every challenging page with me.*

*For Sam and An,
who color every page of this work.*

Abstract

Constraint-based reasoning provides automated decision support across many engineering domains. Examples include constructing the feature model of a software product line, debugging it after a constraint update, and configuring a smartwatch online. As these models grow in size and complexity, inconsistencies and redundancies inevitably arise across the design, maintenance, and usage phases of the model lifecycle, and *diagnostic reasoning* becomes the central capability that explains them and guides their resolution.

The standard machinery for diagnostic reasoning builds on conflict detection and diagnosis algorithms. The dominant cost factor is the number of consistency checks performed by the underlying solver, and minimising solver calls is therefore the central optimisation target. Existing approaches face three fundamental gaps in efficient *conflict detection and diagnosis*, which include the low coverage and instability of pre-computed conflict spaces, the absence of *knowledge reuse* across invocations, and the sequential execution that leaves modern multi-core hardware underutilised. They face three further gaps in the proactive *quality assurance*, which include the absence of test-case-driven *direct diagnosis*, incomplete *redundancy detection*, and the high overhead of checking each test case individually. Together, these gaps waste solver effort across the entire lifecycle of constraint-based models.

In this thesis, we propose six algorithmic contributions that improve diagnostic reasoning in constraint-based systems. The six contributions are organized into two pillars: *Efficient Conflict Detection and Diagnosis* and *Diagnosis-inspired Quality Assurance for Feature Models*. The first pillar, *Efficient Conflict Detection and Diagnosis*, comprises three contributions that advance the core algorithms for identifying conflicts and computing diagnoses. (1) *Robust LCD*, an enhancement to *Lazy Conflict Detection (LCD)*, extracts multiple conflicts from each inconsistent individual and controls genetic diversity through full population reproduction, weighted operators, and adaptive extinction, recovering a substantially larger fraction of the conflict space than the baseline. (2) *INFORMEDQX* introduces knowledge reuse into *QUICKXPLAIN* by storing previously identified conflicts in a lookup table and exploiting them to prune subsequent searches. (3) *FASTDIAGP* parallelises *FASTDIAG* through speculative programming and a configurable look-ahead strategy, pre-computing consistency checks along multiple potential execution paths so that direct diagnosis can exploit the available processor cores.

The second pillar, *Diagnosis-inspired Quality Assurance for Feature Models*, comprises three contributions that extend diagnostic reasoning to QA tasks in feature model engineering. (4) *DIRECTDEBUG* adapts direct diagnosis to test-case-driven debugging and identifies minimal sets of faulty constraints in a

feature model without predetermining conflicts. (5) WIPEOUTR provides complete, solver-independent algorithms for the detection of redundant constraints in feature models and redundant test cases in test suites. (6) AGGREGATEDTEST introduces a divide-and-conquer method that combines compatible test cases into a single consistency check, reducing the number of solver activations required during testing. All six contributions are *solver-agnostic* and interact with the underlying solver only through consistency checks, which keeps them portable across solvers and modeling frameworks.

The proposed techniques have been evaluated on canonical benchmarks that include the *Linux-2.6.33.3* kernel feature model, synthesised models from the Betty generator, and real-world feature models from the SPLOT repository. The results demonstrate substantial improvements over established baselines in conflict coverage, solver-call counts, and runtime, and they show that diagnostic reasoning remains tractable at industrial-scale model sizes. The contributions strengthen the reliability and maintainability of constraint-based systems across the design, maintenance, and usage phases of the lifecycle.

Kurzfassung

Constraint-basiertes Schlussfolgern bietet automatisierte Entscheidungsunterstützung in vielen Fachbereichen des Ingenieurwesens. Beispiele umfassen das Erstellen des Feature-Modells einer Softwareproduktlinie, dessen Debugging nach einer Constraint-Änderung sowie die Online-Konfiguration einer Smart-watch. Mit zunehmender Größe und Komplexität dieser Modelle treten unweigerlich Inkonsistenzen und Redundanzen über die Entwurfs-, Wartungs- und Nutzungsphasen des Modelllebenszyklus hinweg auf. Hierbei wird das *diagnostische Schlussfolgern* zur zentralen Fähigkeit, um diese zu erklären und deren Auflösung anzuleiten.

Die Standardmethode des diagnostischen Schlussfolgerns baut auf Algorithmen zur Konflikterkennung und Diagnose. Der dominierende Kostenfaktor ist hierbei die Anzahl der vom zugrunde liegenden Solver durchgeführten Konsistenzprüfungen, weshalb die Minimierung der Solveraufrufe das zentrale Optimierungsziel darstellt. Bestehende Ansätze weisen drei grundlegende Lücken bei der effizienten *Konflikterkennung und Diagnose* auf. Dazu zählen die geringe Abdeckung und Instabilität vorberechneter Konflikträume, das Fehlen von *Wissenswiederverwendung* über Aufrufe hinweg sowie die sequentielle Ausführung, welche moderne Mehrkernprozessoren unzureichend ausnutzt. Zusätzlich weisen bestehende Ansätze drei weitere Lücken bei der proaktiven *Qualitätssicherung* auf. Zu diesen zählen das Fehlen testfallgetriebener *Direktdiagnose*, unvollständige *Redundanzerkennung* sowie der hohe Aufwand der individuellen Prüfung aller Testfälle. Zusammengefasst führen diese Lücken zu einer Verschwendung von Solveraufwand über den gesamten Lebenszyklus von Constraint-basierten Modellen hinweg.

In dieser Arbeit schlagen wir sechs algorithmische Beiträge vor, die das diagnostische Schlussfolgern in Constraint-basierten Systemen verbessern. Diese Beiträge sind in zwei Schwerpunkte gegliedert: *Effiziente Konflikterkennung und Diagnose* sowie *Diagnose-inspirierte Qualitätssicherung für Feature-Modelle*. Der erste Schwerpunkt, *Effiziente Konflikterkennung und Diagnose*, umfasst drei Beiträge, welche die Kernalgorithmen für die Identifikation von Konflikten und die Berechnung von Diagnosen weiterentwickeln. (1) *Robust LCD*, eine Erweiterung der *Lazy Conflict Detection (LCD)*, extrahiert mehrere Konflikte aus jedem inkonsistenten Individuum und steuert die genetische Diversität durch vollständige Populationsreproduktion, gewichtete Operatoren und adaptive Extinktion, wodurch ein wesentlich größerer Anteil des Konfliktraums als beim Basisverfahren erfasst wird. (2) *INFORMEDQX* führt Wissenswiederverwendung in *QUICKXPLAIN* ein, indem zuvor identifizierte Konflikte in einer Lookup-Tabelle gespeichert und zur Beschneidung nachfolgender Suchen herangezogen werden. (3) *FASTDIAGP* parallelisiert *FASTDIAG* mittels spekulativer Programmierung

und einer konfigurierbaren Look-ahead-Strategie und berechnet Konsistenzprüfungen entlang mehrerer möglicher Ausführungspfade vor, sodass die Direktdiagnose alle verfügbaren Prozessorkerne nutzen kann.

Der zweite Schwerpunkt, *Diagnose-inspirierte Qualitätssicherung für Feature-Modelle*, umfasst drei Beiträge, welche das diagnostische Schlussfolgern für Qualitätssicherungsaufgaben in der Feature-Modell-Entwicklung erweitern. (4) DIRECTDEBUG adaptiert die Direktdiagnose für testfallgetriebenes Debugging und identifiziert minimale Mengen fehlerhafter Constraints in einem Feature-Modell ohne vorherige Konfliktbestimmung. (5) WIPEOUTR stellt vollständige, solverunabhängige Algorithmen zur Erkennung redundanter Constraints in Feature-Modellen und redundanter Testfälle in Testsuiten bereit. (6) AGGREGATEDTEST führt eine Teile-und-herrsche-Methode ein, die kompatible Testfälle zu einer einzigen Konsistenzprüfung zusammenfasst und so die Anzahl der für das Testen erforderlichen Solveraktivierungen reduziert. Alle sechs Beiträge sind *solverunabhängig* und interagieren mit dem zugrunde liegenden Solver ausschließlich über Konsistenzprüfungen. Dadurch wird die Portabilität auf verschiedene Solver und Modellierungs-Frameworks sichergestellt.

Die vorgeschlagenen Techniken wurden auf kanonischen Benchmarks evaluiert, darunter das *Linux-2.6.33.3* Kernel-Feature-Modell, mit dem Betty-Generator synthetisierte Modelle sowie reale Feature-Modelle aus dem SPLOT-Repository. Die Ergebnisse zeigen substanzielle Verbesserungen gegenüber etablierten vergleichbaren Verfahren hinsichtlich Konfliktabdeckung, Anzahl der Solveraufrufe und Laufzeit, und sie belegen, dass diagnostisches Schlussfolgern bei Modellen industrieller Größenordnung handhabbar bleibt. Die Beiträge stärken die Zuverlässigkeit und Wartbarkeit Constraint-basierter Systeme über die Entwurfs-, Wartungs- und Nutzungsphasen des Lebenszyklus hinweg.

Acknowledgements

Pursuing my Ph.D. at the Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, has been a privilege I never took for granted. The years spent on this journey have reshaped my thinking and the questions I find worth asking. This thesis would not exist without the many people who have given me their time, their wisdom, and their trust along the way.

First and foremost, my heartfelt gratitude goes to my supervisor, Professor Alexander Felfernig. I thank him for his constant support throughout this long journey, for the inspiring ideas and research directions he has shared, and for his readiness to comment on every step of my work. I am especially grateful for the collaborative spirit in which we have shaped each contribution together, and for the trust he has placed in the solutions I dared to propose. His guidance has helped me grow as a researcher and find my own scientific voice.

I am also grateful to the second reviewer of this dissertation, Professor Gerhard Friedrich, whose insightful comments have helped me improve this thesis.

Warm thanks go to my colleagues at our institute in Graz: Sebastian Lubos, Tamim Burgstaller, Damian Garber and Mathias Uta. The close cooperation we have built over the years and the many discussions, both technical and otherwise, have left a clear imprint on this thesis.

I am equally indebted to my co-authors on the work behind this thesis. Professor David Benavides, Professor José Galindo, and Cristian Vidal Silva have shaped important parts of this thesis through our shared papers. I would also like to thank Professor Bao Quoc Tang and Dr. Bao-Ngoc Tran for their generous help with the mathematical aspects of the papers behind Chapters 2 and 4. From complexity analysis to the choice of probability functions, their guidance has improved the quality of these contributions.

A special word of thanks goes to my colleagues at the University of Economics, Hue University, in Vietnam, who have created the best possible conditions for me to pursue these doctoral studies abroad. Their encouragement has reached me, even from afar, throughout these years.

My Vietnamese and Austrian friends in Graz have made this city feel like a second home, sharing every joy and every difficulty of these years.

My sister and her husband, my niece and nephew, and all my relatives back home in Vietnam have stood by me without fail at every difficult moment. They have lovingly looked after my parents during the long periods of my absence, and the peace of mind their care has given me cannot be measured. Without it, I could not have focused on this work.

My two sons, Sam and An, are the brightest joy of these years. Their laughter and curiosity have brought light and meaning to a research path full of

challenges. They are, and will always be, the source of my deepest motivation.

No words can fully capture my gratitude to my wife, Trang Tran. She has walked beside me through every step of this work, and her steady encouragement, her patience, and her quiet companionship have carried me through the most difficult periods. Her own example as a researcher has been a daily source of inspiration that I have learned to count on.

Finally, I owe my deepest gratitude to my parents and to my mother-in-law. They have been the emotional anchor of my life and the most luminous example of resilience I have ever known. Their quiet strength runs through every page of this thesis, and through the way I approach every challenge in research and in life.

Viet-Man Le
Graz, May 2026

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date, Signature

Contents

Abstract	v
Kurzfassung	vii
Acknowledgements	ix
Affidavit	xiii
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Background and Motivation	1
1.2 Research Questions	11
1.3 Contributions	12
1.4 Thesis Outline	15
1.5 Publications	16
1.5.1 Publications about this Thesis	16
1.5.2 Publications beyond the Scope of this Thesis	17
2 Robust Lazy Conflict Detection	21
2.1 Introduction	21
2.2 Background and Problem Statement	23
2.2.1 Configuration as a CSP	23
2.2.2 Conflict Sets and Diagnoses	23
2.2.3 Lazy Conflict Detection	24
2.3 Lazy Conflict Detection: Baseline Approach	24
2.3.1 Overview of the Baseline Algorithm	25
2.3.2 Limitations	26
2.4 Enhanced Lazy Conflict Detection	27
2.4.1 Multi-Conflict Extraction	27
2.4.2 Genetic Diversity Control	28
2.5 Empirical Evaluation	30
2.6 Conclusions	35

3	INFORMEDQX	37
3.1	Introduction	37
3.2	Example Configuration Setting	39
3.3	Determining Preferred Minimal Conflicts	41
3.4	INFORMEDQX	46
3.5	Analysis of INFORMEDQX	48
3.6	Conclusions	51
4	FASTDIAGP	53
4.1	Introduction	53
4.2	Related Work	54
4.3	Example Configuration Knowledge Base	55
4.4	Determination of Preferred Diagnoses	56
4.4.1	Preferred Diagnosis	58
4.4.2	FASTDIAG	58
4.5	FASTDIAGP – Parallelized FASTDIAG	59
4.6	Theoretical Analysis of FASTDIAGP	65
4.7	Empirical Evaluation	65
4.8	Conclusions	67
5	DIRECTDEBUG	69
5.1	Introduction	69
5.2	Related Work	70
5.3	Feature Model Semantics	71
5.4	Testing Feature Models	74
5.5	Automated Debugging with DIRECTDEBUG	75
5.6	Performance Analysis	77
5.7	Conclusions	79
6	WIPEOUTR	81
6.1	Introduction	81
6.2	Related Work	82
6.3	Semantics of Feature Models	84
6.4	Redundancy Detection	88
6.5	Performance Analysis	90
6.6	Conclusions and Future Work	91
7	AGGREGATEDTEST	93
7.1	Introduction	93
7.2	Example Feature Model	94
7.3	Feature Model Testing	96
7.4	Determining Aggregated Test Cases	97
7.5	Threats to Validity & Open Issues	100
7.6	Conclusions	101
8	Conclusions and Future Work	103
8.1	Conclusions	103
8.2	Future Work	107
	Bibliography	111

List of Figures

1.1	A <i>presumably faulty</i> feature model of the <i>Smartwatch</i>	8
2.1	Weighted crossover probability vs. parent weight difference	29
2.2	Trade-off between conflict set coverage and solver effort	33
2.3	Convergence of conflict set discovery over generations	34
3.1	QX execution trace	45
3.2	Illustration of <i>Scenario 2.1</i>	47
3.3	Illustration of <i>Scenario 2.2</i>	47
3.4	INFORMEDQX execution trace	49
4.1	FD execution trace	60
4.2	LOOKAHEAD execution trace	63
5.1	A (presumably faulty) <i>survey software</i> feature model	72
5.2	DIRECTDEBUG execution trace	78
6.1	<i>Survey software</i> feature model with a redundant constraint	85
7.1	An example feature model (<i>survey software</i>)	95

List of Tables

1.1	C_{KB} and C_R of <i>Smartwatch</i> configuration task	4
1.2	Overview of thesis contributions	13
2.1	Configuration knowledge bases used in our experiments	30
2.2	Nine configurations	31
2.3	Best configurations per model and metric	32
3.1	Constraints in C_{KB} and C_R	40
3.2	Preference values for conflict sets CS_1 , CS_2 , CS_3	43
3.3	Illustration of <i>lines 6–14</i> in Algorithm 3.3	48
3.4	The <i>complexity</i> of INFORMEDQX and QUICKXPLAIN	50
3.5	Performance of INFORMEDQX vs. QUICKXPLAIN	50
4.1	Constraints in C_{KB} and user requirements C_R	57
4.2	LOOKUP table for the working example	61
4.3	Runtime, speedup, and efficiency of FASTDIAGP vs. FASTDIAG	66
4.4	Average runtime of FASTDIAGP vs. FASTDIAG	67
5.1	Example positive test cases	74
5.2	Example diagnoses	76
5.3	Runtimes of DIRECTDEBUG with varying CF and T_π	77
6.1	Example test cases for the feature model	87
6.2	Trace of WIPEOUTR _T	88
6.3	Trace of WIPEOUTR _{FM}	89
6.4	Characteristics of the Linux-2.6.33.3 feature model	90
6.5	Average runtime of WIPEOUTR _T	91
6.6	Average runtime of WIPEOUTR _{FM}	91
7.1	Feature model constraints C_F	96
7.2	Average runtime of CHOCO on synthesized constraint sets	100
7.3	Average runtime of CHOCO on constraint sets from SPLOT	100

Chapter 1

Introduction

This chapter develops the motivation for the thesis by tracing how diagnostic reasoning challenges arise across the lifecycle of constraint-based systems, from model design through model maintenance to interactive configuration. Section 1.1 introduces the foundational concepts and identifies the limitations of existing approaches. These limitations give rise to six research questions (Section 1.2), each addressed by a specific contribution (Section 1.3). The chapter concludes with the thesis outline (Section 1.4) and a list of publications (Section 1.5).

1.1 Background and Motivation

Constraint-based Reasoning. Many real-world problems require finding solutions that satisfy a set of restrictions simultaneously. Configuring a car involves selecting compatible engine, transmission, and safety options (Mittal and Frayman 1989; Felfernig, Hotz, et al. 2014). Scheduling university courses requires assigning rooms and time slots without conflicts (Castillo et al. 2005; Baptiste et al. 2006). Managing a software product line means ensuring that thousands of feature dependencies remain consistent (Benavides, Segura, and Ruiz-Cortés 2010; Apel, Batory, et al. 2013). These problems share a common structure: they can all be modeled as *constraints* over variables, and solving them amounts to finding assignments that satisfy all constraints at once. This paradigm is known as *Constraint-Based Reasoning (CBR)* (Freuder and Mackworth 1994; Bessiere 2020), a subfield of Artificial Intelligence (Russell and Norvig 2021) closely related to *Knowledge Representation and Reasoning* (Harmelen, Lifschitz, and Porter 2008), which studies how domain knowledge can be formally represented and how automated reasoning can derive conclusions from it.

The formal foundation of CBR is the *Constraint Satisfaction Problem (CSP)* (Tsang 1993; Rossi, Beek, and Walsh 2006). A CSP is defined as a triple (V, D, C) where $V = \{v_1, v_2, \dots, v_n\}$ is a set of variables, $D = \{dom(v_1), dom(v_2), \dots, dom(v_n)\}$ is the set of domains specifying possible values for each variable, and $C = \{c_1, c_2, \dots, c_m\}$ is a set of constraints restricting the allowable combinations of values. A solution to a CSP is a complete assignment of values to all variables such that all constraints are satisfied. *Constraint*

Programming (CP) (Apt 2003; Rossi, Beek, and Walsh 2006) builds upon this formalism and provides a declarative paradigm for modeling and solving combinatorial problems: the modeler specifies variables, domains, and constraints, while the underlying solver employs search and propagation techniques to find solutions. State-of-the-art solvers such as CHOCO (Prud’homme and Fages 2022) have demonstrated that large-scale constraint problems can be solved efficiently in practice. A fundamental operation exposed by such solvers is *consistency checking*, i.e., determining whether a given set of constraints is satisfiable (Felfernig, Reiterer, et al. 2014). Consistency checking thus serves as the primitive on which all techniques discussed in this thesis are built.

Constraint-based systems (CBSs) embed constraint solvers into software tools that provide automated reasoning and decision support. For example, in *knowledge-based configuration*, CBSs help customers assemble valid product configurations subject to technical and business constraints (Stumptner 1997; Junker 2006; Felfernig, Hotz, et al. 2014). In configuring a smartwatch, for instance, a CBS can ensure that a high resolution screen is not paired with a battery too small to power it. Similar systems have been deployed across high-stakes industrial domains, from automotive and telecommunications to railway interlocking systems (Fleischanderl et al. 1998; Falkner and Fleischanderl 2001; Sinz, Kaiser, and Kuchlin 2003). In *constraint-based recommender systems*, user preferences are integrated with domain constraints to generate personalized suggestions, such as recommending travel packages that match both the user’s budget and airline availability (Felfernig and Burke 2008). In *Software Product Lines (SPLs)*, CBSs reason over feature models to support *configuration tasks* (deriving valid products from user requirements) as well as *analysis operations* such as dead feature detection (Trinidad et al. 2008; White et al. 2010; Benavides, Segura, and Ruiz-Cortés 2010). These reasoning tasks become especially challenging at industrial scale, where the underlying models contain thousands of variables and constraints. Delivering such systems therefore requires systematic attention across the entire *lifecycle* of a constraint-based model, from its initial construction by knowledge engineers, through its ongoing maintenance, to its interactive use by end users (Fleischanderl 2002).

At every stage of this lifecycle, constraints can conflict with one another or with user inputs, and some can become redundant. Detecting such *conflicts* and *redundancies*, locating their causes, and explaining them to the affected stakeholder are the central tasks of *diagnostic reasoning* (Reiter 1987; Felfernig, Friedrich, Jannach, et al. 2004). Since each phase of the lifecycle requires a different form of diagnostic reasoning, we describe the three phases first, before unpacking its concepts and approaches.

Phases of the Constraint-based Model Lifecycle. This lifecycle unfolds across three main phases: *design*, *maintenance*, and *usage*. In each phase, the corrective action takes the form of an *explanation* that identifies the underlying cause and guides the resolution (Gupta, Genc, and O’Sullivan 2021; Felfernig, Falkner, and Benavides 2024).

In the *design phase*, knowledge engineers construct the constraint model that encodes domain knowledge. Building a configuration model for a smartwatch product line, for instance, requires translating hardware compatibility rules, software dependencies, and marketing constraints into formal constraints.

This process is inherently error-prone: cognitive overload, incomplete domain knowledge, and *outdated rules* frequently introduce two kinds of defects (Zeller 2001; Trinidad et al. 2008; Benavides, Felfernig, et al. 2013). Some constraints contradict one another and produce an inconsistent model, while others are logically implied by existing constraints and contribute nothing new. The central challenge of this phase is therefore to detect and eliminate *inconsistencies* and *redundancies* before the model is released to downstream phases (Zeller 2001; Benavides, Felfernig, et al. 2013).

In the *maintenance phase*, the constructed model is validated, debugged, and kept consistent as it evolves (Fleischanderl 2002; Felfernig, Friedrich, Jannach, et al. 2004). Validation relies on test cases that specify the intended and unintended behaviors of the model. For instance, a test case may stipulate that a smartwatch with a high resolution screen and a 100 mAh battery should not be a valid solution. Over time, new variables and constraints are introduced while existing ones become obsolete, which can reintroduce both conflicts and redundancies (Sabin and Freuder 1999). After each such change, the test cases are re-evaluated, and any *unsatisfied test case* reveals that the model wrongly rejects a solution it should accept, or wrongly accepts one it should reject. These re-evaluations, however, become expensive as both model size and test suite size grow (Shchekotykhin, Friedrich, et al. 2012). The central challenge of this phase is therefore to efficiently identify *unsatisfied test cases*, localize faulty constraints, and remove newly introduced redundancies at the combinatorial scale of models and test suites (Felfernig, Friedrich, Jannach, et al. 2004; Benavides, Felfernig, et al. 2013).

In the *usage phase*, the deployed CBS is in productive use. End users interact with it by specifying their requirements, selecting features, or providing preferences, depending on the application (McGuinness 2007; Felfernig and Burke 2008). A customer, for instance, may request a smartwatch with a high resolution screen and a 100 mAh battery, only to have the request rejected. Whenever such user inputs are incompatible with the background knowledge, the system must respond with an explanation of the conflict so that the user can revise their selection. Interactive responsiveness is essential at this stage. In online configurators and recommender systems, response times of more than a few seconds can lead to user abandonment (Nielsen 1994). The central challenge of this phase is therefore to compute fast, user-relevant *explanations* for *unsatisfied user inputs* under tight interactive latency budgets (O’Sullivan et al. 2007).

All three phases face a cross-cutting challenge of *scalability*. Modern knowledge bases contain thousands of variables and constraints, solver calls are computationally expensive, and models evolve over time as new variables or constraints are introduced. Keeping constraints up to date is itself a moving target that can reintroduce faults over time (Nieke et al. 2018). All three phases thus demand efficient diagnostic reasoning, which is the central motivation of this thesis.

Diagnostic Reasoning: Concepts and Approaches. The three phase-challenges outlined above share a common formal shape: a set of constraints that ought to be jointly satisfiable turns out not to be. Diagnostic reasoning resolves such situations by producing explanations that reveal *why* the constraints are inconsistent and *how* to restore consistency (Gupta, Genc, and O’Sullivan

1.1. Background and Motivation

Table 1.1: C_{KB} and C_R of *Smartwatch* configuration task.

CSP Representation	
Domain-specific Constraints (C_{KB})	
c_1	$\neg(\text{Screen} = \text{HiRes} \wedge \text{Battery} = 100\text{mAh})$
c_2	$\text{Camera} \neq N \rightarrow \text{Screen} = \text{HiRes}$
c_3	$\text{Compass} = t \rightarrow \text{GPS} = t$
User Requirements (C_R)	
c_4	$\text{Screen} = \text{HiRes}$
c_5	$\text{Battery} = 100\text{mAh}$
c_6	$\text{Compass} = t$
c_7	$\text{GPS} = f$

2021). It comprises two foundational tasks: *conflict detection*, which identifies what *causes* the inconsistency, and *diagnosis computation*, which identifies what must be *removed* to restore consistency (Junker 2004; Reiter 1987; Felfernig, Schubert, and Zehentner 2012).

To reason about these two tasks uniformly across phases, the constraints of a CSP (V, D, C) are partitioned into a *background set* $B \subseteq C$ of constraints treated as correct and not subject to retraction, and a *consideration set* $C^* = C \setminus B$ of potentially faulty constraints (Junker 2004; Felfernig, Schubert, and Zehentner 2012; Gupta, Genc, and O’Sullivan 2021). This partition specializes to each phase: at *design time*, B holds constraints considered correct and C^* contains constraints being debugged; at *maintenance time*, B holds correct constraints together with a test suite and C^* contains constraints being analyzed; at *usage time*, B holds the domain-specific constraints C_{KB} and C^* contains the user requirements C_R .

To illustrate the diagnostic reasoning concepts introduced in this thesis, we use a running example of an *over-constrained configuration task* from the domain of *Smartwatch configuration*. The configuration task is formulated as a CSP (V, D, C) (see Example 1.1). A solution of such a CSP is called a *configuration*, i.e., a complete assignment that satisfies all constraints in C . We refer to (Hotz et al. 2014) for further details on the configuration task.

Example 1.1 (*Smartwatch* configuration task). A CSP-based *Smartwatch* configuration task (V, D, C) is the following:

- $V = \{\text{Screen}, \text{Battery}, \text{Camera}, \text{Compass}, \text{GPS}\}$
- $\text{dom}(\text{Screen}) = \{\text{Analog}, \text{HiRes}\}$, $\text{dom}(\text{Battery}) = \{100\text{mAh}, 200\text{mAh}, 300\text{mAh}\}$, $\text{dom}(\text{Camera}) = \{N, 3\text{MP}, 5\text{MP}\}$, $\text{dom}(\text{Compass}) = \{(t)\text{rue}, (f)\text{alse}\}$, $\text{dom}(\text{GPS}) = \{t, f\}$
- $C = C_{KB} \cup C_R$, where $C_{KB} = \{c_1, c_2, c_3\}$ and $C_R = \{c_4, c_5, c_6, c_7\}$ (see Table 1.1 for constraints in C_{KB} and C_R).

Some of the user requirements C_R are inconsistent with the constraints in C_{KB} . For instance, c_4 and c_5 together require a high resolution screen with a 100 mAh battery, which violates c_1 . Similarly, c_6 requires a compass, which via c_3 implies $\text{GPS} = t$, contradicting c_7 . Therefore, no solution can be found for this configuration task.

Such over-constrained tasks motivate the two foundational tasks of diagnostic reasoning: pinpointing *which* constraints conflict and proposing *which* ones

to relax, as developed in the remainder of this section. In terms of the partition (B, C^*) introduced above, this example instantiates the usage-time case with $B = C_{KB}$ and $C^* = C_R$.

Conflict Detection. To explain why C^* is inconsistent with B , we look for a subset of C^* that is itself inconsistent with B . Formally, a *conflict set* $CS \subseteq C^*$ is a subset of the consideration set that is inconsistent with the background, i.e., $\text{inconsistent}(B \cup CS)$. A conflict set is *minimal* (i.e., *subset-minimal*) if no proper subset of it is also a conflict set (Junker 2004). Minimal conflict sets are also known in the *Boolean Satisfiability (SAT)/CP* literature as *minimal unsatisfiable subsets (MUSs)* (Liffiton and Sakallah 2008; Gregoire, Mazure, and Piette 2008). The most prominent algorithm for conflict detection is QUICKXPLAIN (Junker 2004), which employs a divide-and-conquer strategy to efficiently isolate a *minimal conflict set*. Given a consideration set of size n with c elements in the conflict, QUICKXPLAIN requires at most $2c \cdot \log_2(n/c) + 2c$ consistency checks in the worst case (Junker 2004). Each invocation returns a single minimal conflict. Enumerating all minimal conflicts requires wrapping it in the *Hitting Set Directed Acyclic Graph (HSDAG)* discussed below (Reiter 1987; Greiner, Smith, and Wilkerson 1989). Due to its simplicity and formal guarantees of minimality, QUICKXPLAIN is widely adopted (Rodler 2022) and is conceptually related to *Delta Debugging* algorithm (Zeller and Hildebrandt 2002) from software engineering, which applies the same divide-and-conquer idea to isolate minimal failure-inducing test inputs. Several variants have been proposed to optimise QUICKXPLAIN, including speculative execution that anticipates consistency checks along potential branches and runs them in parallel to reduce wall-clock time (Vidal et al. 2021).

For the *Smartwatch* configuration task in Example 1.1, the two minimal conflict sets are: $CS_1 = \{c_4, c_5\}$, where a high resolution screen paired with a 100 mAh battery violates c_1 , and $CS_2 = \{c_6, c_7\}$, where requesting a compass while disabling GPS violates the constraint c_3 . Neither set contains a proper subset that remains inconsistent with C_{KB} .

Diagnosis. Users typically require not just knowledge of conflicts but actionable suggestions on how to restore consistency. A *diagnosis* $\Delta \subseteq C^*$ is a minimal subset of the consideration set whose removal restores consistency, i.e., $\text{consistent}(C^* \setminus \Delta \cup B)$ (Reiter 1987; de Kleer and Williams 1987). In the SAT/CP literature, diagnoses are also known as *minimal correction subsets (MCSs)* (Marques-Silva, Heras, et al. 2013). The complement of a diagnosis, $C^* \setminus \Delta$, is known as a *Maximal Satisfiable Subset (MSS)* (Liffiton and Sakallah 2008), which is a largest subset of the consideration set that remains consistent with B .

There are two main approaches to computing diagnoses: *conflict-driven diagnosis* and *direct diagnosis*. The conflict-driven approach, represented by the HSDAG algorithm (Reiter 1987; Greiner, Smith, and Wilkerson 1989), systematically enumerates diagnoses by repeatedly invoking a conflict detection algorithm such as QUICKXPLAIN (Junker 2004). In this paradigm, diagnoses correspond to *minimal hitting sets* of all conflict sets (Reiter 1987), which is why conflict-driven diagnosis is also referred to as *hitting-set-based diagnosis*. HSDAG invokes QUICKXPLAIN up to d times per diagnosis, where d is the di-

agnosis size. In the worst case, this amounts to $d \cdot (2c \cdot \log_2(n/c) + 2c)$ consistency checks per diagnosis, and the total cost of enumerating all minimal diagnoses scales linearly with their number. This approach yields both explanations (conflict sets tell the user *why* the configuration fails) and repairs (diagnoses tell the user *how* to fix it). The direct approach bypasses conflicts and computes diagnoses directly via divide-and-conquer on the constraint set, and its representative algorithm is FASTDIAG (Felfernig, Schubert, and Zehentner 2012). Given n constraints with a diagnosis of size d , FASTDIAG requires at most $2d \cdot \log_2(n/d) + 2d$ consistency checks per invocation. Each invocation returns a single minimal diagnosis. Enumerating all minimal diagnoses requires repeated invocations that exclude previously-found diagnoses, analogous to how HSDAG wraps QUICKXPLAIN. Direct diagnosis is more efficient when only diagnoses are needed and conflict sets are not required as explanations.

For the *Smartwatch* configuration task in Example 1.1, applying HSDAG to the conflicts $CS_1 = \{c_4, c_5\}$ and $CS_2 = \{c_6, c_7\}$ yields four minimal diagnoses: $\Delta_1 = \{c_4, c_6\}$, $\Delta_2 = \{c_4, c_7\}$, $\Delta_3 = \{c_5, c_6\}$, and $\Delta_4 = \{c_5, c_7\}$. For instance, $\Delta_1 = \{c_4, c_6\}$ suggests that the user could resolve the inconsistency by accepting an analog screen and dropping the compass requirement, which preserves the desired 100 mAh battery and the choice to disable GPS.

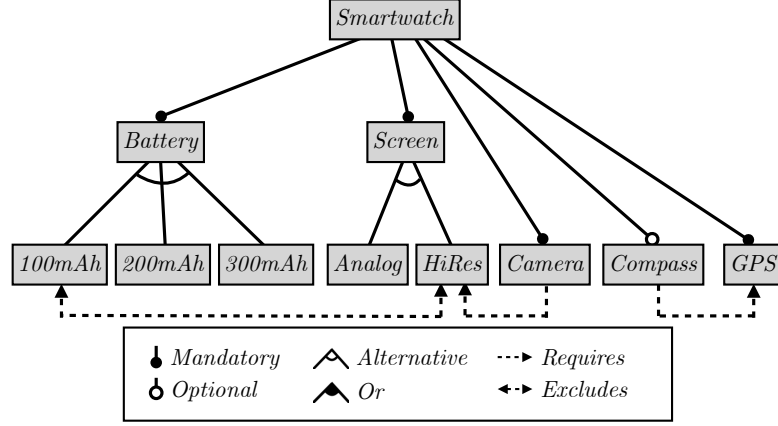
Preferred Conflicts and Diagnoses. In interactive settings such as on-line configurators and recommender systems, users expect near-real-time responses (Nielsen 1994). Consequently, conflicts and diagnoses must be computed within milliseconds. Enumerating all conflicts or all diagnoses is typically infeasible, and users are usually satisfied with one well-chosen result (O’Sullivan et al. 2007; Walsh 2007; Rossi, Venable, and Walsh 2011). To this end, both QUICKXPLAIN and FASTDIAG accept a *strict total order* over the constraints in C^* that encodes user preferences, ranking constraints from most to least important (Junker 2004; Marques-Silva and Previti 2014). Given such an ordering, the *preferred minimal conflict set* CS_P is the unique conflict containing the constraints the user is most willing to relax. Dually, the *preferred minimal diagnosis* Δ_P is the unique diagnosis that suggests adapting those same least-important constraints first. For example, in the *Smartwatch* task, suppose the user insists on the high resolution screen (c_4) but is flexible about battery capacity (c_5). The preferred conflict is $CS_2 = \{c_6, c_7\}$, which avoids blaming the screen, rather than $CS_1 = \{c_4, c_5\}$. The preferred diagnosis is $\Delta_3 = \{c_5, c_6\}$ or $\Delta_4 = \{c_5, c_7\}$, which relax the battery, rather than Δ_1 or Δ_2 , which would drop the screen.

Solver-Agnostic Approaches. QUICKXPLAIN, HSDAG, and FASTDIAG are *solver-agnostic*, a unifying design property they share. These methods treat the underlying solver as a black box and rely solely on consistency checks over carefully chosen subsets of constraints. By contrast, *solver-specific* techniques (e.g., SAT clause-learning (Marques-Silva, Lynce, and Malik 2009) or conflict-directed backjumping (Prosser 1993)) are tied to a single solver and difficult to port (Gupta, Genc, and O’Sullivan 2021). The solver-agnostic design is particularly valuable in domains such as feature modeling and product configuration, where the underlying reasoner may vary across deployments. Accordingly, all algorithms developed in this thesis belong to this family.

Lazy Conflict Detection. While QUICKXPLAIN efficiently identifies one minimal conflict per run, it remains impractical for large-scale configuration knowledge bases because each invocation incurs substantial solver overhead, and conflicts must often be recomputed after each minor change in user requirements during interactive configuration (Vidal et al. 2021). *Lazy Conflict Detection (LCD)* (Uran and Felfernig 2018) addresses this scalability issue through *knowledge reuse*: minimal conflict sets are predetermined offline and stored in a lookup table, so that runtime requests are served by direct retrieval rather than solver calls. Building on this idea, Uran and Felfernig propose an LCD algorithm based on a genetic algorithm that explores candidate user requirement sets, discovering as many conflict sets as possible to populate the lookup table. Each individual in the population encodes a set of user requirements. If the individual is inconsistent with the domain knowledge base, a conflict set is extracted using QUICKXPLAIN and added to the table. The discovered conflict then guides reproduction through a conflict-driven resolve step, which derives candidate parents by relaxing one constraint at a time from the conflict. Crossover and mutation then yield the next generation, and the loop continues until a time budget is exhausted.

Open Challenges. In summary, diagnostic reasoning encompasses conflict detection and diagnosis computation, both driven by repeated consistency checks against the underlying solver. Although each individual check can be answered efficiently by a modern solver, the overall cost of diagnostic reasoning is dominated by the *number* of consistency checks performed across the entire computation. Minimizing solver calls is therefore the central optimization target for diagnostic reasoning. Despite the effectiveness of existing approaches, they face fundamental limitations along this axis that motivate the first three contributions of this thesis:

1. **Low coverage and weak robustness of LCD** ($\rightarrow$ Chapter 2): LCD’s baseline genetic algorithm suffers from three critical limitations as a pre-computation strategy. First, only 20–30% of all minimal conflict sets are typically discovered, because each inconsistent individual contributes only one conflict, while it may in fact carry several. Second, the evolutionary process converges prematurely (Katoch, Chauhan, and Kumar 2021), leaving large regions of the conflict space unexplored. Third, repeated executions yield inconsistent results, undermining reproducibility. Together, these limitations waste the offline precomputation budget LCD relies on.
2. **No knowledge reuse in conflict detection** ($\rightarrow$ Chapter 3): QUICKXPLAIN computes from scratch every time, without exploiting previously identified conflicts. In interactive configuration sessions, where a user iteratively refines requirements, similar constraint sets recur across invocations. Each new call to QUICKXPLAIN repeats consistency checks that could have been avoided, wasting solver effort across invocations.
3. **Sequential execution in direct diagnosis** ($\rightarrow$ Chapter 4): FASTDIAG’s consistency checks execute sequentially even on modern multi-core hardware, because the algorithm’s branching decisions depend on previous results. While prior work parallelized constraint solving (Bordeaux, Hamadi,

Figure 1.1: A *presumably faulty* feature model of the *Smartwatch*.

and Samulowitz 2009; Gent et al. 2018) and hitting-set computation (Jan-nach, Schmitz, and Shchekotykhin 2016), direct diagnosis was not similarly addressed. On large-scale knowledge bases such as the Linux kernel feature model (6,467 features, 13,972 constraints), consistency checking becomes a severe bottleneck, wasting available CPU cores that could otherwise speed up diagnosis.

Feature Models as a Motivating Domain. The remaining three contributions of this thesis are developed in the context of *Software Product Lines*, a particularly important application domain for diagnostic reasoning. An SPL manages a family of related software products that share commonalities while exhibiting controlled variability. *Feature Models (FMs)* were introduced by the *Feature-Oriented Domain Analysis (FODA)* method (Kang et al. 1990) in the form now known as *basic feature models*, which capture hierarchical relationships together with cross-tree constraints. They have since become the *de facto* standard for representing variability in SPLs (She et al. 2010; Galindo, Benavides, and Segura 2010; Rodas-Silva et al. 2019; Benavides, Sundermann, et al. 2025). More advanced variants such as *cardinality-based* feature models (Czarnecki, Helsen, and Eisenecker 2005) and *extended* feature models (Batory 2005) additionally support cardinalities greater than one and attribute-valued feature descriptions, respectively. The constructions in this thesis focus on basic feature models. A *presumably faulty Smartwatch* feature model is depicted in Figure 1.1 and serves as the running example throughout this section.

In a basic feature model, the hierarchy is typically a rooted tree in which parent-child relationships fall into four standard types: *mandatory* (e.g., *Battery* under *Smartwatch* in Figure 1.1), *optional* (e.g., *Compass*), *alternative* (e.g., exactly one of *Analog* or *HiRes* for the *Screen*), and *or* (at least one child of an or-group must be selected). Cross-tree constraints additionally capture dependencies and exclusions between features located in different parts of the tree, such as the *requires* from *Camera* to *HiRes* and the *excludes* between *HiRes* and *100mAh*.

Feature models can be transformed into CSPs by mapping each feature to

a Boolean variable (selected or not selected) and by translating both the hierarchical relationships and the cross-tree constraints into logical constraints. A complete set of rules for translating an FM into its constraint-based representation is given by Benavides et al. (Benavides, Segura, and Ruiz-Cortés 2010). This transformation enables *diagnostic reasoning* as well as *analysis operations* on FMs.

Feature Model Analysis Operations. Feature model analysis operations can be regarded as the computer-aided analysis of feature model properties (Benavides, Segura, and Ruiz-Cortés 2010; Galindo, Benavides, Trinidad, et al. 2019). Around 30 such operations have been catalogued, ranging from consistency checking and configuration counting to anomaly detection, redundancy analysis, and explanation generation. Example analysis operations include detecting *void feature models* (models that admit no valid configuration), *dead features* (features that cannot be selected in any valid configuration), *false optional features* (features included in all configurations although not modeled as mandatory), *full mandatory features* (features included in every valid configuration), and *redundant constraints* (constraints whose removal does not change the set of valid configurations). The *Smartwatch* FM in Figure 1.1 exhibits several such defects: the *requires* from *Camera* to *HiRes* together with the mandatory *Camera* forces *HiRes* to always be selected, making *HiRes* both false optional and full mandatory while leaving *Analog* and *100mAh* dead. In addition, the *requires* from *Compass* to *GPS* is redundant since *GPS* is already mandatory.

Performing these operations requires one or more solver calls per feature or constraint (Trinidad et al. 2008; Benavides, Segura, and Ruiz-Cortés 2010; Kowal, Ananieva, and Thüm 2016), making feature model *quality assurance (QA)* computationally expensive. In practice, this cost is amplified by the scale of real-world feature models. The *Linux-2.6.33.3* kernel feature model, for instance, contains 6,467 features and 13,972 constraints (She et al. 2010; Heradio et al. 2022). Moreover, FMs continuously evolve, and the addition of new features, constraints, or test cases can easily introduce contradictions that make parts of the model unsatisfiable. FMs thus serve as an ideal motivating domain. They are complex enough to expose the limitations of existing diagnostic techniques, while remaining representative of broader issues in CBR. Although FMs are the primary motivating domain for the diagnostic techniques developed in this thesis, these techniques are designed to apply broadly to any CBS.

Diagnosis-inspired Quality Assurance. Diagnostic reasoning has so far focused on resolving inconsistencies. The same machinery, however, extends to *proactive quality assurance*, which detects and eliminates model defects before users encounter them. Such proactive QA occurs during the design and maintenance phases of the model, and takes three forms. First, *inconsistencies* between the model and *test cases*, which encode its intended semantics, can be detected and diagnosed (Felfernig, Friedrich, Jannach, et al. 2004). Second, *redundant constraints* form a distinct class of defect. They leave the solution space unchanged but inflate maintenance and solver cost (Sabin and Freuder 1999). Third, repeated test-suite execution introduces a scalability concern as models evolve (Fraser and Wotawa 2007).

Test-case-driven diagnosis. A conflict-driven diagnosis approach using *test cases* that specify intended and unintended behavior of constraint-based knowledge bases was originally proposed in (Felfernig, Friedrich, Jannach, et al. 2004). An example of such test cases in the context of the *Smartwatch* feature model (Figure 1.1) is $t_1 : \text{Analog} = \text{true}$, which requires *Analog* to be selected in some valid configuration. Another is $t_2 : \text{HiRes} = \text{false} \vee 100\text{mAh} = \text{false}$, which forbids the selection of both *HiRes* and *100mAh* in the same configuration. Test case t_1 is *violated*, i.e., $\text{inconsistent}(\{t_1\} \cup C_F \cup \{c_0\})$, since the mandatory *Camera* requires *HiRes*, leaving *Analog* dead. Here C_F is the *Smartwatch* FM’s constraint set, and $c_0 : \text{Smartwatch} = \text{true}$ is the root constraint that prevents the derivation of empty configurations. Test case t_2 , by contrast, is satisfied because the *excludes* between *HiRes* and *100mAh* already rules out their joint selection. Such violated test cases *induce conflicts* in the model, for which diagnosis identifies a minimal $\Delta \subseteq C_F$ whose removal restores consistency.

Redundancy detection. A second concern is the detection of *redundant constraints* in FMs (Trinidad et al. 2008; Felfernig, Benavides, et al. 2013). A constraint $c \in C$ is *redundant* if it is logically implied by the remaining constraints, i.e., $C \setminus \{c\} \models c$, so that removing c leaves the set of valid configurations unchanged. The same notion extends to test suites T : a test case $t \in T$ is *redundant* if $t' \models t$ for some other test case $t' \in T$, since t' already entails t . Redundancies of both kinds accumulate as models and test suites evolve during maintenance, eroding model understandability and increasing the cost of solver-based reasoning.

Test-suite scalability. A third concern is the cost of repeated test-suite execution (Fraser and Wotawa 2007; Mendonca, Wąsowski, and Czarnecki 2009). As feature models evolve, regression-style validation re-executes the test suite, and a one-test-per-call schedule becomes a scalability bottleneck for industrial-scale FMs. Two test cases $t_i, t_j \in T$ are *compatible* when their conjunction is consistent with the model, i.e., $\text{consistent}(\{t_i, t_j\} \cup C_F)$, opening the door to combined evaluation and reduced solver-call cost.

Open QA Challenges. Proactive QA in this sense aims to ensure that models remain efficient and comprehensible over time. The same QA-relevant concerns (inconsistencies, redundancies, and scalability bottlenecks) arise in product configurators, recommender systems, and other CBSs where the same diagnostic-reasoning machinery applies. The remaining three contributions of this thesis target three open QA challenges along the same solver-call axis:

4. **No test-case-driven direct diagnosis** ($\rightarrow$ Chapter 5): Test-case-driven diagnosis was originally addressed via conflict-driven approaches (Felfernig, Friedrich, Jannach, et al. 2004) that first detect conflicts and then derive diagnoses via HSDAG. Direct diagnosis avoids this pipeline but had not been adapted to integrate test suites. Test-case-driven FM diagnosis therefore remained tied to the conflict-driven pipeline, wasting solver calls on conflict pre-determination.
5. **Incomplete redundancy detection** ($\rightarrow$ Chapter 6): No complete algorithms existed for detecting redundant constraints in feature models

or redundant test cases in FM test suites. Existing approaches either missed redundancies (Piette 2008; Sabin and Freuder 1999) or were limited to specific solver contexts such as linear programming (Paulraj and Sumathi 2010; Fukada, Gärtner, and Szedlak 2015), lacking a general, solver-independent framework and wasting maintenance effort on artifacts that could be eliminated.

6. **Individual test case checking overhead** ($\rightarrow$ Chapter 7): Without a mechanism to aggregate compatible test cases, each test case requires a separate consistency check (Fraser and Wotawa 2007). For large test suites, this becomes a severe scalability bottleneck. No prior method exploited the compatibility structure among test cases, wasting solver activations that could have been batched together.

Motivation for the Thesis. Across all phases of the constraint-based model lifecycle, efficient diagnostic reasoning is indispensable. The six limitations identified above span these phases: limitations 1–3 target conflict detection and diagnosis at usage time, while limitations 4–6 target QA activities at design and maintenance time.

This thesis addresses all six limitations, bridging the gap between theoretical advances and industrial requirements. The next section formalizes them as research questions.

1.2 Research Questions

Building on the limitations identified in the previous section, this thesis formulates six research questions, each corresponding to one thesis chapter (see also Table 1.2).

Topic 1: Pre-computation of Conflict Sets. The three limitations of baseline LCD discussed earlier (low coverage, premature convergence, and result instability) motivate a robust enhancement that addresses all three simultaneously.

RQ1: *How can lazy conflict detection be made more robust and efficient through multi-conflict extraction and diversity control mechanisms?*

$\rightarrow$ Addressed in Chapter 2.

Topic 2: Knowledge Reuse in Online Conflict Detection. To address the lack of knowledge reuse in QUICKXPLAIN, a reuse-aware variant could store previously identified conflicts and exploit them across invocations in interactive configuration sessions.

RQ2: *How can knowledge from previously detected conflicts be systematically reused to improve the efficiency of conflict detection?*

$\rightarrow$ Addressed in Chapter 3.

Topic 3: Parallelization of Direct Diagnosis. To exploit modern multi-core hardware, the sequential execution of FASTDIAG could be parallelized via *speculative programming* (Burton 1985), where computations along multiple execution paths are launched in parallel before the actual path is known.

RQ3: *How can parallelization and speculative computation be integrated into direct diagnosis to reduce runtime on multi-core systems?*

→ Addressed in Chapter 4.

Topic 4: Automated Testing and Debugging of Feature Models. To extend direct diagnosis to test-case-driven FM debugging, a method must integrate test suites at the meta-level to identify minimal sets of faulty constraints without predetermining conflicts.

RQ4: *How can direct diagnosis techniques be adapted to support automated testing and debugging of feature models?*

→ Addressed in Chapter 5.

Topic 5: Redundancy Detection in Feature Models and Test Suites. To detect redundancies in both FM constraints and test suites, a complete, solver-independent algorithmic framework is needed for both artifact types.

RQ5: *How can redundant constraints in feature models and redundant test cases in test suites be automatically identified?*

→ Addressed in Chapter 6.

Topic 6: Efficient Test Case Aggregation. To exploit the compatibility structure among test cases, an aggregation method is needed that determines which compatible tests can be combined into a single consistency check.

RQ6: *How can test cases be aggregated to minimize the number of consistency checks required for feature model testing?*

→ Addressed in Chapter 7.

1.3 Contributions

Table 1.2 provides an overview of the six main contributions of this thesis, mapping each research question to its corresponding contribution and thesis chapter. All six contributions are entirely *solver-agnostic*. They interact with the solver only through consistency checks and therefore remain portable across constraint solvers and modeling frameworks (Gupta, Genc, and O’Sullivan 2021).

The six contributions are organized around two thematic pillars. The first pillar, *Efficient Conflict Detection and Diagnosis* (RQ1–RQ3), advances the core algorithms for identifying conflicts and computing diagnoses in CBSs: Robust LCD improves the robustness and coverage of offline conflict pre-computation, INFORMEDQX introduces knowledge reuse into online conflict detection, and

Table 1.2: Overview of thesis contributions.

RQ	Contribution	Publication	Ch.
RQ1	Robust LCD	Le, Feldgrill, Felfernig (AAAI 2026)	2
RQ2	INFORMEDQX	Le, Felfernig, Tran, Uta (AAAI 2024)	3
RQ3	FASTDIAGP	Le, Vidal Silva, Felfernig et al. (AAAI 2023)	4
RQ4	DIRECTDEBUG	Le, Felfernig, Uta et al. (ICSE-NIER 2021)	5
RQ5	WIPEOUTR	Le, Felfernig, Uta et al. (SPLC 2022)	6
RQ6	AGGREGATEDTEST	Le, Felfernig, Tran (SPLC 2022)	7

FASTDIAGP parallelizes direct diagnosis for multi-core architectures. The second pillar, *Diagnosis-inspired Quality Assurance for Feature Models* (RQ4–RQ6), extends diagnostic reasoning to QA tasks in feature model engineering: DIRECTDEBUG extends direct diagnosis to test-case-driven debugging of feature models, WIPEOUTR detects redundancies in both feature model constraints and test suites, and AGGREGATEDTEST aggregates compatible test cases for efficient feature model testing. In the following, we describe each contribution in detail.

RQ1 – Robust LCD (Chapter 2). We propose a robust enhancement to LCD, a genetic algorithm-based method for pre-computing minimal conflict sets offline. Our approach addresses three critical limitations of the baseline LCD: low conflict coverage, premature convergence, and result instability. We introduce two complementary enhancement strategies. First, *multi-conflict extraction* integrates HSDAG (Reiter 1987) with QUICKXPLAIN (Junker 2004) to discover multiple minimal conflict sets from a single inconsistent individual, rather than extracting only one. Second, *genetic diversity control* prevents premature convergence through three mechanisms: full population reproduction, weighted genetic operators that bias parent selection toward conflict-rich individuals, and adaptive extinction that resets stagnant populations. Empirical evaluations on five real-world configuration knowledge bases demonstrate that our approach recovers up to 85% of known conflict sets, reduces solver calls by up to 73%, and achieves substantially higher result stability compared to the baseline (Le, Feldgrill, and Felfernig 2026).

RQ2 – INFORMEDQX (Chapter 3). We introduce INFORMEDQX, a reuse-aware variant of QUICKXPLAIN (Junker 2004) for conflict detection in CBSs. While standard QUICKXPLAIN treats each activation as an independent computation, INFORMEDQX stores previously identified conflict sets in a lookup table and exploits them to accelerate subsequent conflict detection sessions. The algorithm uses a bitmap-based preference function to rank conflict sets according to user-defined priorities and prunes the search space by recognizing when a previously identified conflict is still present in the current constraint set. A theoretical analysis establishes the correctness and complexity properties of the algorithm. Empirical evaluations on the *Linux-2.6.33.3* configuration knowledge base show runtime reductions of up to 45% and full elimination of constraint solver calls (from 22 to 0 in the best case) when the preferred conflict can be retrieved from

the lookup table, compared to standard QUICKXPLAIN (Le, Felfernig, Tran, and Uta 2024).

RQ3 – FASTDIAGP (Chapter 4). We present FASTDIAGP, a parallelized version of FASTDIAG (Felfernig, Schubert, and Zehentner 2012) for direct diagnosis in CBSs. FASTDIAGP adopts the principle of speculative programming (Burton 1985) to pre-calculate consistency checks that may be requested by the sequential algorithm. A configurable *look-ahead* strategy launches computations along multiple possible execution paths before the actual path is known. This mechanism enables FASTDIAGP to exploit modern multi-core architectures, providing consistency checks with fast answers and scaling performance with the number of available processor cores. Empirical evaluations on the *Linux-2.6.33.3* configuration knowledge base show runtime improvements of up to 40% with 8 CPU cores. A theoretical analysis of the algorithm’s complexity complements the empirical findings (Le, Vidal Silva, et al. 2023).

RQ4 – DIRECTDEBUG (Chapter 5). We introduce DIRECTDEBUG, a direct diagnosis approach to the automated testing and debugging of FMs. DIRECTDEBUG extends the direct diagnosis paradigm to integrate test cases as central elements of the diagnosis process. Given a feature model and a set of test cases that specify the intended and unintended behavior of the model, DIRECTDEBUG identifies minimal sets of faulty constraints responsible for inconsistencies between the test cases and the feature model, without predetermining conflict sets. Empirical evaluations on synthesised FMs from the Betty generator (Segura et al. 2012) show runtimes in the millisecond range for moderate model sizes (e.g., approximately 39 msec for $|CF| = 100$ and $|T_\pi| = 100$) and confirm tractability up to $|CF| = 1,000$ constraints with $|T_\pi| = 500$ test cases (under six seconds in the largest tested setting) (Le, Felfernig, Uta, Benavides, et al. 2021).

RQ5 – WIPEOUTR (Chapter 6). We present WIPEOUTR, an algorithmic approach for the automated identification of redundancies in feature models and feature model test suites. We introduce two algorithms: (1) WIPEOUTR_{FM}, which identifies a maximum set of constraints that can be removed from an FM without changing its semantics (i.e., the set of valid configurations), and (2) WIPEOUTR_T, which detects test cases in a test suite whose removal does not reduce fault detection capability. A performance analysis with the *Linux-2.6.33.3* feature model reveals a redundancy degree of 34.36% in the original model (4,801 out of 13,972 constraints are redundant). The evaluation also demonstrates that removing redundant constraints improves both configuration search runtime and testing efficiency (Le, Felfernig, Uta, Tran, et al. 2022).

RQ6 – AGGREGATEDTEST (Chapter 7). We introduce a divide-and-conquer approach for aggregating feature model test cases to reduce the number of solver calls required during testing. Our approach transforms a test suite into an aggregated representation where compatible test cases, whose conjunction remains consistent with the feature model, are checked together in a single solver call. We also provides a best- and worst-case runtime analysis of the aggregation algorithm. Evaluations on both synthesised FMs (using the Betty generator

(Segura et al. 2012)) and real-world FMs (from the SPLOT repository (Mendonça, Wąsowski, and Czarnecki 2009)) show that AGGREGATEDTEST reduces the number of solver activations by up to a factor of 12 (from $|T| = 60$ individual checks to $|TA| = 5$ aggregated checks) and the corresponding solver runtime by up to 88% on the largest tested SPLOT model (Le, Felfernig, and Tran 2022).

Implementation and availability. All six algorithms are publicly available as open-source reference implementations linked in their respective chapters. They are complemented by two dedicated tools listed in Section 1.5.2, namely the DIRECTDEBUG software package (Le, Felfernig, Tran, Atas, et al. 2021) and the FMTESTING plug-in for FeatureIDE (Burgstaller et al. 2023). Three of the algorithms (FASTDIAGP, DIRECTDEBUG, and WIPEOUTR) have additionally been integrated as plugins into the `flamapy` framework¹ (Galindo, Horcas, et al. 2023), an open-source Python ecosystem for the automated analysis of feature models. FASTDIAGP is also accessible interactively through the browser-based environment `flamapy.ide`² (Benitez et al. 2025).

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapters 2–4 address the first pillar of this work, focusing on efficient conflict detection and diagnosis. Chapters 5–7 form the second pillar, applying diagnosis-inspired techniques to QA for feature models.

In **Chapter 2**, we address the limitations of the baseline LCD. We first formalize the configuration task as a CSP and describe the baseline approach. We then present two complementary enhancement strategies: multi-conflict extraction, which integrates HSDAG with QUICKXPLAIN to discover multiple conflicts per individual, and genetic diversity control, which prevents premature convergence through full population reproduction, weighted genetic operators, and adaptive extinction. An empirical evaluation on five real-world configuration knowledge bases demonstrates improvements in coverage, efficiency, and robustness.

Chapter 3 introduces a knowledge reuse strategy for conflict detection in the context of constraint-based configuration. Using a working example from the smartwatch domain, we formalize the notion of preferred minimal conflicts and revisit the QUICKXPLAIN algorithm. Based on this, we present INFORMEDQX, which exploits previously identified conflicts to decide whether conflict detection needs to be activated or whether a preferred conflict has already been determined. A theoretical analysis and an empirical evaluation on the *Linux-2.6.33.3* configuration knowledge base demonstrate the correctness and efficiency gains of the approach.

In **Chapter 4**, we turn to the parallelization of direct diagnosis. After reviewing related work on solution search, conflict detection, and conflict resolution, we introduce an example configuration knowledge base and formalize the determination of preferred diagnoses. We then present FASTDIAGP, a parallelized version of FASTDIAG that adopts the speculative programming principle to pre-calculate consistency checks potentially required by the algorithm. A

¹<https://www.flamapy.org>

²<https://ide.flamapy.org>

theoretical analysis and an empirical evaluation on the *Linux-2.6.33.3* configuration knowledge base demonstrate runtime improvements scaling with the number of available processor cores.

The second part of the thesis shifts focus to feature model QA. **Chapter 5** first reviews related work and introduces the semantics of FMs as CSPs. We then define the notion of test cases for feature models and present DIRECTDEBUG, a direct diagnosis approach to automated testing and debugging. DIRECTDEBUG identifies faulty constraints responsible for an unintended behavior of a feature model without predetermining conflicts, using test cases as input. A performance analysis compares DIRECTDEBUG against traditional hitting set-based debugging approaches.

Chapter 6 addresses the problem of redundancy in feature models and test suites. After reviewing related work on feature model analysis and redundancy identification, we formalize feature model semantics and present two algorithms: WIPEOUTR_{FM} for detecting redundant constraints in feature models and WIPEOUTR_T for identifying redundant test cases in test suites. A performance analysis with a real-world feature model evaluates the runtime improvements of both algorithms.

Chapter 7 focuses on improving the efficiency of feature model testing through test case aggregation. We introduce an example feature model from the survey software domain and define the testing framework. We then present a divide-and-conquer approach that transforms a test suite into an aggregated representation where individual test cases can be combined if specific consistency criteria are fulfilled. The chapter also presents a best- and worst-case runtime analysis and discusses threats to validity and open issues.

Finally, **Chapter 8** summarizes the contributions of this thesis, discusses limitations, and outlines directions for future research.

1.5 Publications

All contributions in this thesis have been published in peer-reviewed conference/workshop proceedings. Each chapter begins by stating the underlying publication and the author’s specific contributions. In the following, we provide a complete list of publications, clearly separating the core thesis publications from additional contributions.

1.5.1 Publications about this Thesis

The following six publications form the basis of Chapters 2–7. They are listed in chapter order.

1. **[Chapter 2]** V.M. Le, L.A. Feldgrill, and A. Felfernig (Mar. 2026). “Robust Lazy Conflict Detection via Multi-Conflict Extraction and Genetic Diversity Control”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 40.23, pp. 19208–19215
2. **[Chapter 3]** V.M. Le, A. Felfernig, T.N.T. Tran, and M. Uta (Mar. 2024). “INFORMEDQX: Informed Conflict Detection for Over-Constrained Problems”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.9, pp. 10616–10623

3. **[Chapter 4]** V.M. Le, C. Vidal Silva, et al. (June 2023). “FASTDIAGP: An Algorithm for Parallelized Direct Diagnosis”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37.5, pp. 6442–6449
4. **[Chapter 5]** V.M. Le, A. Felfernig, M. Uta, D. Benavides, et al. (2021). “DIRECTDEBUG: Automated Testing and Debugging of Feature Models”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 81–85
5. **[Chapter 6]** V.M. Le, A. Felfernig, M. Uta, T.N.T. Tran, et al. (2022). “WIPEOUTR: automated redundancy detection for feature models”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A. SPLC '22*. Graz, Austria: Association for Computing Machinery, pp. 164–169
6. **[Chapter 7]** V.M. Le, A. Felfernig, and T.N.T. Tran (2022). “Test case aggregation for efficient feature model testing”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B. SPLC '22*. Graz, Austria: Association for Computing Machinery, pp. 174–177

1.5.2 Publications beyond the Scope of this Thesis

Besides the core publications listed above, the author contributed to additional work in the areas of knowledge representation and configuration for feature models, tool support for feature model testing and debugging, and new directions including causality-based explanations and the application of Large Language Models (LLMs) to SPLs. The following publications are related to the thesis topics but are not included as thesis chapters. They are grouped thematically.

Foundational and Supporting Techniques.

7. A. Felfernig, V.M. Le, and T.N.T. Tran (2020). “Supporting Feature Model-Based Configuration in Microsoft Excel”. In: *22nd International Configuration Workshop, (ConfWS' 20)*, pp. 35–38
→ Proposes an approach to represent feature model configuration knowledge in Microsoft Excel, making configuration accessible to non-IT stakeholders and supporting corrective explanations for configuration inconsistencies.
8. A. Felfernig, B. Ortner, and V.M. Le (2022). “Table-Based Knowledge Representations for Industrial Feature Models”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B. SPLC '22*. Association for Computing Machinery, pp. 245–248
→ Presents systematic translations of feature model relationships and cross-tree constraints into standardized table-based representations, enabling domain experts in industrial settings to define and exchange configuration knowledge without formal knowledge representation backgrounds.
9. A. Felfernig, V.M. Le, and S. Lubos (2023). “Conjunctive Query Based Constraint Solving for Feature Model Configuration”. In: *Proceedings*

of the 12th Conference on Information Technology and Its Applications. Vol. 734. Springer Nature Switzerland, pp. 357–367

→ Introduces a conjunctive query based approach to feature model configuration that leverages relational databases as an alternative to specialized constraint solvers such as CSP, SAT, and ASP, lowering the technology integration barrier for organizations with existing SQL-based infrastructure.

Tool Support and Software.

10. V.M. Le, A. Felfernig, T.N.T. Tran, Müslüm Atas, et al. (2021). “DIRECT-DEBUG: A software package for the automated testing and debugging of feature models”. In: *Software Impacts* 9, p. 100085

→ Provides an open-source Java implementation of the DIRECTDEBUG framework for automated testing and debugging of variability models. The implementation is based on the concepts and algorithm presented in Chapter 5.

11. T. Burgstaller et al. (2023). “FMTESTING: A FEATUREIDE Plug-in for Automated Feature Model Analysis and Diagnosis”. In: *Proceedings of the 26th European Conference on Artificial Intelligence*. Vol. 372. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 3190–3195

→ Extends FeatureIDE, an Eclipse plugin for feature-oriented software development, with feature model testing and debugging capabilities, providing developers with minimal repair suggestions for detected anomalies directly within their modeling environment. The tool builds on the concepts and algorithms presented in Chapters 5, 6, and 7.

12. V.M. Le, T.N.T. Tran, and A. Felfernig (2021). “A Conversion of Feature Models into an Executable Representation in Microsoft Excel”. In: *Intelligent Systems in Industrial Applications*. Springer International Publishing, pp. 153–168

→ Presents a tool (FM2EXCONF) that converts FMs into executable Excel-based configurators, supporting anomaly detection and configuration explanation.

Extensions and New Directions.

13. A. Felfernig, D. Garber, et al. (2025). “Causality-based Explanations for Feature Model Configuration”. In: *Proceedings of the 19th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, pp. 86–90

→ Introduces the CXPLAIN algorithm for computing minimal causality-based explanations of feature model configurations. The algorithm addresses the question of why specific features were automatically included or excluded by the configurator, enabling transparency in the configuration process.

14. V.M. Le, T.N.T. Tran, S. Lubos, et al. (2026). “Early-Stage Product Line Validation Using LLMs: A Study on Semi-Formal Blueprint Analysis”. In: *Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing*. ACM, pp. 1620–1627

→ Proposes an early validation workflow for SPL scoping that uses LLMs to perform feature model analysis operations directly on semi-formal textual blueprints, providing early feedback on consistency and feasibility at the pre-formal scoping phase. By showing that LLMs can reason accurately about feature model semantics, the work opens a path toward using them to translate the conflict sets and diagnoses produced by the algorithms in this thesis into natural language and to explain why these outputs arise, making them accessible to end users, a direction discussed under *Explainability of diagnostic and QA outputs* in the Future Work of Chapter 8.

Chapter 2

Robust Lazy Conflict Detection via Multi-Conflict Extraction and Genetic Diversity Control

This chapter is based on the results documented in (Le, Feldgrill, and Felfernig 2026). The author of this thesis proposed the core idea, contributed to the algorithm design, and carried out the baseline implementation, empirical evaluation, data analysis, and writing of the paper.

Detecting minimal conflict sets is essential for providing meaningful feedback in knowledge-based configuration. While *lazy conflict detection* addresses runtime efficiency by predetermining conflict sets offline using a genetic algorithm, it suffers from low conflict coverage, stagnation, and instability. We propose a robust enhancement that integrates *multi-conflict extraction* and *genetic diversity control* to overcome these limitations.¹ Our method extends conflict discovery per evaluation and introduces three diversity mechanisms: full population reproduction, weighted genetic operators, and adaptive extinction. Empirical evaluations on five real-world configuration knowledge bases show that our approach recovers up to 85% of conflict sets, reduces solver calls by up to 73%, and achieves higher result stability. These improvements demonstrate the scalability and reliability of enhanced lazy conflict detection for interactive configuration systems.

2.1 Introduction

Detecting minimal conflict sets (MCSs), also known as minimal unsatisfiable subsets (MUSs), is a central task in knowledge-based configuration (Mittal and Frayman 1989; Fleischanderl et al. 1998; Felfernig, Reiterer, et al. 2014). When user requirements lead to inconsistency, identifying a minimal explana-

¹The work presented in this chapter has been developed within the scope of the GENRE project (Generative AI for Requirements Engineering), which is funded by the Austrian Research Promotion Agency (915086).

tion is essential for interactive systems to provide meaningful and actionable feedback (Nielsen 1994; Junker 2006). Existing approaches, such as QUICKXPLAIN (Junker 2004), efficiently identify one conflict set per run but remain impractical for large-scale configuration knowledge bases (CKBs) due to their high solver overhead (O’Sullivan et al. 2007; Jannach, Schmitz, and Shchekotykhin 2015; Vidal et al. 2021; Le, Vidal Silva, et al. 2023).

To address this scalability issue, *lazy conflict detection* (LCD) (Uran and Felfernig 2018) proposes an offline strategy: minimal conflict sets are predetermined using a genetic algorithm and stored in a lookup table for real-time retrieval during configuration. While this significantly reduces solver calls at runtime (Le, Felfernig, Tran, and Uta 2024), the approach suffers from several critical limitations: (1) **coverage gaps**, where only 20–30% of MCSs are discovered; (2) **stagnation**, where the evolutionary process prematurely converges, limiting further exploration; and (3) **instability**, where repeated executions yield inconsistent outcomes.

These limitations arise from well-known challenges in evolutionary search, such as limited exploration, dependency on the initial population, and unbalanced selection pressure (Mitchell 1998; Katoch, Chauhan, and Kumar 2021). Nevertheless, prior work has not systematically addressed them in the context of conflict detection (Bessiere et al. 2013; Carneiro, Novais, and Neves 2013).

In this chapter, we address these issues via four complementary enhancements, organized into two strategic directions:

- **Multi-Conflict Extraction:** Enhances conflict discovery by extracting multiple minimal conflict sets from each individual in the population. This is achieved by applying the HSDAG algorithm (Hitting Set Directed Acyclic Graph) (Reiter 1987; Greiner, Smith, and Wilkerson 1989) in combination with QUICKXPLAIN to systematically enumerate MCSs associated with a given configuration.
- **Genetic Diversity Control**, which comprises three mechanisms:
 - *Full population reproduction*, including consistent individuals in crossover;
 - *Weighted genetic operators*, prioritizing parents more likely to yield conflicts;
 - *Adaptive extinction*, that resets the population upon detecting stagnation.

We empirically evaluate these mechanisms, both individually and in combination, on five real-world CKBs ranging from compact models (Arcade) to large-scale benchmarks (EA). Our results show that the proposed method (1) achieves up to **85%** conflict set coverage, significantly outperforming the baseline; (2) reduces solver calls by up to **73%** through more efficient conflict discovery; (3) improves result stability, with variance reductions of up to **50%** over multiple independent runs; and (4) generalizes effectively across CKBs of varying size and constraint complexity.

Overall, this work demonstrates that combining multi-conflict extraction with diversity-aware genetic search yields a scalable and robust framework for offline conflict predetermination in configuration systems.

2.2 Background and Problem Statement

Knowledge-based configuration tasks are commonly modeled as *constraint satisfaction problems* (CSPs) (Rossi, Beek, and Walsh 2006), where the objective is to find a valid assignment to a set of variables subject to domain constraints and user-defined requirements.

2.2.1 Configuration as a CSP

A *configuration task* and its *configuration* (solution) are defined as follows:

Definition 2.1 (Configuration task and Configuration knowledge base). A configuration task can be defined as a CSP (V, D, C) where (1) $V = \{v_1 \dots v_n\}$ is a set of finite domain variables, (2) $D = \{dom(v_i) : v_i \in V\}$ is a set of domain definitions for each variable $v_i \in V$, and (3) $C = C_{KB} \cup C_R$ is a set of constraints restricting possible solutions of a configuration task. C_{KB} represents a set of domain-specific constraints, and C_R represents a set of user requirements. Finally, (V, D, C_{KB}) is denoted as a configuration knowledge base.

Definition 2.2 (Configuration). A configuration (solution) S for a given configuration task (V, D, C) is an *assignment* $A = \{v_1 = a_1 \dots v_n = a_n\}$, $a_i \in dom(v_i)$. S is *valid* if it is *complete* (each variable in V has a value) and *consistent* (S fulfills the constraints in C).

When $C_R \cup C_{KB}$ is inconsistent, no valid configuration exists. In such situations, we are interested in explanations as to why no solution could be identified. MCSs, also known as MUSs, are a means often used to explain such inconsistent situations (O’Sullivan et al. 2007; Bendík and Černá 2020). In the following, we formally define *minimal conflict sets* and discuss how to identify them to support the resolution of inconsistent configuration tasks.

2.2.2 Conflict Sets and Diagnoses

Conflict sets are constraint sets that are responsible for an inconsistency, i.e., a situation in which no solution can be found. We use $consistent(C)$ to indicate a consistent constraint set C , and $inconsistent(C)$ for situations without a solution for C .

Definition 2.3 (Conflict set). A *conflict set* is a set $CS \subseteq C_R : inconsistent(CS \cup C_{KB})$. CS is *minimal* iff $\nexists CS' \subset CS : inconsistent(CS' \cup C_{KB})$.

Definition 2.3 introduces the concept of conflict set minimality on the basis of *subset minimality* (i.e., not minimal cardinality). This means if CS is a minimal conflict, no proper subset of CS is a minimal conflict. With conflict set minimality, each conflict can be resolved by removing one element from the corresponding conflict set.

One of the most efficient and widely adopted algorithms to conflict detection is QUICKXPLAIN (Rodler 2022). This algorithm was introduced by Junker (Junker 2004), which supports the determination of a minimal conflict set in a given set of constraints C . Following a divide-and-conquer search regime, the underlying idea of QUICKXPLAIN is to divide the consideration set into two subsets. If the first subset is inconsistent, then conflict detection should be

2.3. Lazy Conflict Detection: Baseline Approach

applied to this subset. The second subset must not be further analyzed since at least one conflict exists in the first subset. This way, the consideration set can be reduced by half with a single consistency check.

QUICKXPLAIN determines exactly one conflict set per computation. In order to identify all minimal conflicts, this algorithm has to be combined with the HSDAG-based diagnosis algorithm (Reiter 1987; Greiner, Smith, and Wilkerson 1989). HSDAG repeatedly activates a conflict detection algorithm (e.g., QUICKXPLAIN) and analyzes different possibilities to resolve the returned conflict. This helps to identify all minimal conflict sets and corresponding minimal diagnoses in the consideration set C (Felfernig, Schubert, and Zehentner 2012; Nica et al. 2013). For further details of the algorithm, we refer to (Reiter 1987) and (Greiner, Smith, and Wilkerson 1989).

To resolve all conflicts, we need to identify corresponding hitting sets, also known as diagnoses (Reiter 1987), that have to be adapted or deleted to make the user requirements consistent with the knowledge base. Based on the definition of a conflict set, we now introduce the definition of a *diagnosis task* and a corresponding *diagnosis*.

Definition 2.4 (Diagnosis task). A diagnosis task for a configuration task (V, D, C) can be defined by a tuple (C_R, C_{KB}) , where C_R is a set of user requirements to be analyzed and C_{KB} is a set of constraints specifying the configuration knowledge base assumed to be consistent.

Definition 2.5 (Diagnosis). A *diagnosis* Δ of a diagnosis task (C_R, C_{KB}) is a set $\Delta \subseteq C_R : \text{consistent}(C_R \setminus \Delta \cup C_{KB})$. Δ is *minimal* iff $\nexists \Delta' \subset \Delta : \text{consistent}(C_R \setminus \Delta' \cup C_{KB})$.

2.2.3 Lazy Conflict Detection

Lazy conflict detection is an offline strategy designed to reduce the number of solver calls during runtime in knowledge-based configuration. Instead of identifying minimal conflict sets on demand, the algorithm precomputes a set of conflicts by simulating user requirement scenarios using a genetic algorithm (GA). Each individual in the GA represents a set of user requirements. If an individual is inconsistent with the CKB, a conflict set is extracted using QUICKXPLAIN. Discovered conflicts are stored in a lookup table for real-time reuse during interactive sessions (Le, Felfernig, Tran, and Uta 2024).

This precomputation strategy significantly improves runtime responsiveness and enables scalable conflict detection (Le, Felfernig, Tran, and Uta 2024). However, the baseline method faces several limitations, including low coverage of MCSs, population stagnation, and inconsistent results across runs. These weaknesses primarily stem from premature convergence in the GA and the limited conflict information extracted per iteration. We address these limitations in the following sections by introducing enhancements that improve both the effectiveness and stability of lazy conflict detection.

2.3 Lazy Conflict Detection: Baseline Approach

LCD, originally introduced by Uran and Felfernig (2018), enables precomputation of conflict sets using a genetic algorithm, thereby reducing the runtime cost

Algorithm 2.1 Lazy Conflict Detection (Baseline)

```

1:  $Population \leftarrow InitializePopulation(N, P_0)$ 
2:  $ConflictBase \leftarrow \emptyset$ 
3: while time limit not reached do
4:    $Parents \leftarrow \emptyset$ 
5:   for each  $C_R \in Population$  do
6:     if  $inconsistent(C_{KB} \cup C_R)$  then
7:        $CS \leftarrow QuickXPlain(C_R, C_{KB})$ 
8:        $ConflictBase \leftarrow ConflictBase \cup \{CS\}$ 
9:        $Parents \leftarrow Parents \cup Resolve(C_R, CS)$ 
10:    end if
11:  end for
12:   $NewGen \leftarrow Crossover(Parents)$ 
13:   $Population \leftarrow Mutate(NewGen, P_m)$ 
14: end while
15: return  $ConflictBase$ 

```

of online conflict detection. In this section, we present the baseline algorithm and highlight its core limitations.

2.3.1 Overview of the Baseline Algorithm

The baseline approach uses a GA to explore sets of user requirements. Each individual represents a set of user requirements C_R . The overall process (see Algorithm 2.1) consists of repeated evaluation and evolution steps until a time or iteration budget is reached.

Initialization. The algorithm begins by generating an initial population of N individuals. Each individual is formed by randomly assigning values to user requirements, with a probability P_0 that a requirement remains unassigned (*line 1*). This encourages diversity in the starting population, mixing consistent and inconsistent sets of user requirements.

Evaluation. Each individual is evaluated for consistency with the configuration knowledge base C_{KB} (*line 6*). If an individual is found to be inconsistent, the algorithm invokes QUICKXPLAIN to extract a minimal conflict set (*line 7*), which is then stored in the global conflict base (*line 8*). The resulting conflict set is also forwarded to the *Resolve* function, which generates candidate parents for the next generation (*line 9*).

The *Resolve* function plays a central role in conflict-driven exploration. Given an inconsistent individual C_R and its associated conflict set $CS \subseteq C_R$, it generates a set of modified individuals by removing one constraint at a time from CS . These variants represent minimal repairs, each corresponding to a potential diagnosis.

Formally, for each $c \in CS$, a new candidate is created as $C'_R = C_R \setminus \{c\}$. All such resolved variants, regardless of whether they are consistent, are added to the parent pool. Notably, the original inconsistent individual is excluded from

reproduction. This strategy ensures that genetic material is shaped by minimal adaptations of known conflicts, encouraging diversity without redundant rediscovery of previously analyzed inconsistencies.

Reproduction. The reproduction step generates a new population by applying the genetic operators, crossover and mutation, to the parent pool constructed during the evaluation phase (*lines 12–13*).

Crossover. Pairs of parents are randomly selected from the set of resolved individuals. For each pair, a new offspring is generated by recombining their user requirement assignments. Recombination is performed by iterating over the union of user requirement variables that appear in either parent, ensuring that no relevant assignment is omitted. The value of each variable in the offspring is determined as follows:

- If both parents assign the same value to a variable, the offspring inherits that value.
- If one parent assigns a value and the other leaves it unassigned, the offspring inherits the assigned value with probability 0.5.
- If the parents assign different values, the offspring randomly adopts one of the two.

This crossover strategy preserves structural traits from both parents while introducing variation in the offspring population. The process is repeated until the new population reaches the target size N .

Mutation. After crossover, mutation is applied to each variable in the offspring with probability P_m . If a variable is selected for mutation, it is assigned a new value randomly drawn from its domain, regardless of whether it was previously assigned. This mutation mechanism promotes exploration by introducing new set of user requirements and helps prevent premature convergence.

The baseline algorithm does not incorporate any explicit fitness function or ranking mechanism. As a result, the only form of selection pressure arises from the conflict-driven repair process that determines which individuals are eligible to become parents.

At the end of each iteration, the newly generated and mutated offspring replace the current population for the next generation. The algorithm then repeats this process iteratively until the predefined time limit is reached (*line 3*).

2.3.2 Limitations

While the baseline algorithm offers a scalable alternative to online conflict detection, it exhibits three major limitations that affect its practical effectiveness.

First, the algorithm often suffers from *coverage gaps*, discovering only a small fraction (typically 20–30%) of the minimal conflict sets. This issue stems from the fact that each inconsistent individual yields at most one conflict set via QUICKXPLAIN, which limits the diversity of conflicts that can be extracted in each generation.

Second, the search process is vulnerable to *premature convergence*. Due to the absence of an explicit fitness function or diversity preservation mechanisms,

the population tends to converge quickly toward regions of the search space that produce conflicts early, thereby reducing the likelihood of exploring alternative conflict-inducing user requirements. This stagnation often leads to redundant rediscovery of similar conflict sets across generations.

Third, the algorithm exhibits *instability across runs*. Because the GA relies on random initialization and stochastic operators, different executions may yield substantially different sets of conflicts. Without dedicated control over population diversity, the outcome is sensitive to initial conditions and may not be repeatable.

These limitations arise from structural characteristics of the baseline GA: single-conflict extraction per individual, absence of explicit diversity preservation, and random recombination without guidance. To overcome these challenges, we introduce several enhancements aimed at increasing conflict coverage, reducing solver calls, and improving stability. These enhancements are presented in the next section.

2.4 Enhanced Lazy Conflict Detection

To address the limitations of the baseline, we introduce four enhancements grouped into two strategic directions: Multi-Conflict Extraction and Genetic Diversity Control. These mechanisms are designed to increase conflict coverage, reduce solver effort, and improve robustness across runs.

2.4.1 Multi-Conflict Extraction

The baseline algorithm applies QUICKXPLAIN to extract a single minimal conflict set per inconsistent individual (*line 7* in Algorithm 2.1). This introduces a critical limitation: if an individual contains multiple conflict sets, only one of them is identified per individual. The idea is that other conflict sets will emerge in later generations due to variation introduced by *Resolve*. However, this is not guaranteed. In fact, the genetic operators, crossover and mutate, may remove constraints involved in yet-undetected conflicts, leading to permanent loss of valuable conflict information.

To address this issue, we enhance the evaluation step by integrating QUICKXPLAIN with the HSDAG algorithm (Reiter 1987; Greiner, Smith, and Wilkerson 1989). This integration enables the discovery of multiple minimal conflict sets from a single inconsistent individual. Specifically, HSDAG orchestrates repeated invocations of QUICKXPLAIN to systematically enumerate conflict sets within the given user requirements. This enhancement increases the likelihood of capturing the complete set of conflicts before the user requirements is altered through genetic operators.

The conflict identification process terminates once a predefined limit on the number of extracted conflict sets (e.g., 10) is reached. This cap is essential to balance the benefits of broader conflict discovery with the computational cost of the conflict identification process.

Overall, this enhancement improves the conflict coverage rate, particularly for large-scale or densely constrained knowledge bases, where multiple conflicts per individual are common. By discovering several conflicts in a single evaluation step, the algorithm reduces reliance on stochastic discovery and prevents

premature loss of critical conflict sets.

2.4.2 Genetic Diversity Control

To address the issues of premature convergence and instability, we introduce three complementary mechanisms that reshape the genetic search dynamics by promoting diversity and strategically guiding selection.

Full Population Reproduction. The baseline algorithm restricts the parent pool to resolved variants derived from inconsistent individuals. In contrast, this strategy incorporates both consistent individuals and resolved variants into the parent pool. By broadening the diversity of traits available for recombination, this approach enhances exploratory potential and mitigates the risk of stagnation in conflict-prone regions.

Weighted Genetic Operators. To further guide the genetic search while maintaining population diversity, we introduce a set of *weighted genetic operators*. These mechanisms dynamically adjust the likelihood of selection and influence inheritance dynamics, prioritizing individuals that previously contributed to conflict discovery.

Weighted Conflicts. This mechanism biases parent selection toward individuals that contribute further conflict sets. During the evaluation phase of LCD, inconsistent individuals are analyzed to identify multiple minimal conflict sets. Each resulting *resolved variant*, created by removing a diagnosis that resolves all identified conflict sets, is assigned a weight proportional to the number of conflicts detected in the original individual.

Formally, if an individual yields k minimal conflict sets and l corresponding diagnoses, then each of the l resolved variants receives a weight of k , capturing the conflict density of its origin. In contrast, consistent individuals are assigned a base weight of zero. The probability of selecting a resolved variant i for crossover is defined as:

$$P(i) = \frac{1 + w_i}{\sum_j (1 + w_j)}$$

where w_i is the number of conflicts identified in the individual i . This prioritization directs the search toward areas likely to yield additional conflicts while still allowing occasional selection of consistent individuals for exploration.

Conflict Redundancy Avoidance (CRA). Since multiple resolved variants can stem from the same inconsistent individual, weighted selection may inadvertently select closely related parents, reducing genetic diversity. To prevent this, we introduce *Conflict Redundancy Avoidance*.

Each group of resolved individuals derived from a single inconsistent origin is assigned a unique identifier. During crossover, the algorithm ensures that no two selected parents share the same origin ID. This safeguard reduces the chance of producing offspring that replicate known conflicts, thereby enhancing coverage of previously unexplored regions in the search space.

Weighted Crossover. Beyond influencing which parents are selected, we also bias the crossover process based on parental weights. Let $\Delta_w = w_1 - w_2$ denote the weight difference between two selected parents. The probability that the

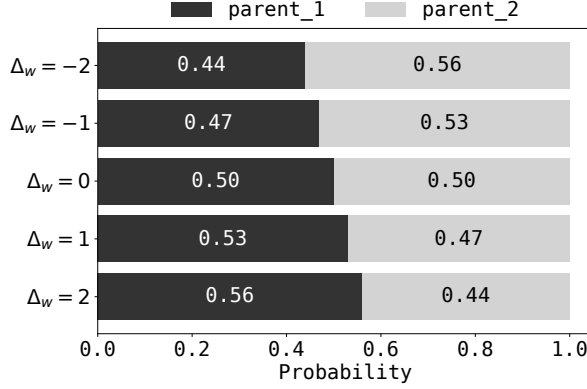


Figure 2.1: Weighted crossover probability as a function of parent weight difference (Δ_w), with $c = 2$. Positive Δ_w favors parent 1; negative Δ_w favors parent 2.

offspring inherits a trait from the higher-weight parent is computed using the following smoothed function:

$$probability = \arctan\left(\frac{\Delta_w}{\pi^c}\right) \cdot 0.3 + 0.5$$

Here, c is a tunable constant that controls the steepness of the curve, and we set $c = 2$ in our implementation. As shown in Figure 2.1, the function yields a probability of 0.5 when both parents have equal weights ($\Delta_w = 0$), ensuring uniform inheritance. As the absolute value of Δ_w increases, the probability gradually shifts toward the higher-weight parent, up to a maximum of approximately 0.8. This smooth, bounded adjustment ensures that promising traits are more likely to be passed on without fully excluding the other parent.

This mechanism helps reinforce successful conflict-yielding patterns while avoiding excessive exploitation. Combined with CRA, it supports a balance between intensification and diversification.

Together, these weighted genetic operators, including conflict-based weighting, redundancy-aware pairing, and weight-sensitive recombination, enhance the efficiency, stability, and robustness of the genetic search. They enable the algorithm to concentrate solver effort on promising areas of the user requirement space while maintaining the diversity necessary to uncover new conflicts.

Adaptive Extinction. A well-known issue in genetic algorithms is the tendency toward convergence, especially in aging populations. Over time, genetic diversity diminishes as dominant traits proliferate, reducing the algorithm’s ability to explore new regions of the search space. This problem is particularly acute in smaller populations or in domains with sparse conflict distribution, where the search may become trapped in local basins of attraction.

To counteract this stagnation, we implement an *adaptive extinction* mechanism that monitors the algorithm’s progress and resets the population when exploration stalls. Specifically, if a predefined number of generations pass without discovering any new minimal conflict sets, the entire population is discarded

2.5. Empirical Evaluation

Table 2.1: Configuration knowledge bases (CKBs) used for our experiments (Arcade=Arcade Game PL, FQA=Feature model for Functional Quality Attributes, B2C=Feature model for a B2C system with fixed priced products, BusyBox=Feature model for the software suite BusyBox, and EA=Feature model for EA 2468). $|\mathbb{N}|$ represents the number of known minimal conflict sets, while the symbol $\sim$ indicates that additional minimal conflict sets may exist.

CKB	Arcade	FQA	B2C	BusyBox	EA
$ V $	47	101	194	683	1,069
$ C_{KB} $	66	101	233	405	2,670
$ \mathbb{N} $	142	366	267	161	$\sim 8,981$

and replaced with a randomly initialized one. This *soft reset* rejuvenates the search process, allowing the algorithm to escape local optima and re-enter unexplored regions of the user requirement space.

The extinction threshold, i.e., the number of stagnant generations tolerated before extinction is triggered, is configurable and can be tailored to the complexity of the configuration knowledge base. In practice, this threshold is chosen conservatively to avoid premature resets while still responding to genuine stagnation.

Extinction can also be combined with a global termination criterion that halts the algorithm after a fixed number of extinction cycles. When used alongside the *Weighted Conflicts* mechanism, adaptive extinction plays a complementary role: the weighting scheme drives the search toward conflict-rich areas of the search space, which are eventually exhausted, triggering extinction and encouraging exploration of structurally different conflict regions.

By introducing periodic resets conditioned on search stagnation, adaptive extinction preserves long-term diversity, improves coverage of the conflict set space, and prevents overfitting to known conflicts. It is particularly effective in large or heterogeneous CKBs, where conflicts may be scattered across diverse regions of the user requirement space.

2.5 Empirical Evaluation

We evaluate the effectiveness, efficiency, and stability of our enhancements to *lazy conflict detection* across five real-world configuration knowledge bases (CKBs) collected from the S.P.L.O.T. feature model repository (Mendonca, Wąsowski, and Czarnecki 2009) and the Diverso Lab’s benchmark (Heradio et al. 2022). These models (see Table 2.1) span a range of sizes and constraint densities, from compact domains like Arcade to large-scale industrial models such as EA. Our goal is to assess three key dimensions: conflict set coverage, solver efficiency, and run-to-run stability, with detailed results summarized in Table 2.3 and trade-off/convergence plots shown in Figures 2.2 and 2.3. Complete results, extended analyses, and full plots are provided in the Technical Appendix², while the implementation and evaluation artifacts are publicly available³.

²Appendix: <https://doi.org/10.5281/zenodo.17610291>

³Artifact: <https://github.com/AIG-ist-tugraz/robust-lcd>

Table 2.2: Nine configurations.

Configuration	Multi-CS Extraction	Full Population Reproduction	Weighted Conflicts	Conflict Avoidance	Redundancy	Weighted Crossover	Adaptive Extinction
<i>Baseline</i>							
<i>Multi-CS</i>	✓						
<i>Full Pop</i>	✓	✓					
<i>Weighted Basic</i>	✓	✓	✓				
<i>Weighted Advanced</i>	✓	✓	✓	✓			
<i>Weighted Full</i>	✓	✓	✓	✓		✓	
<i>Extinction</i>							✓
<i>Ext+Multi-CS</i>	✓						✓
<i>All Features</i>	✓	✓	✓	✓		✓	✓

Table 2.3: Best configurations per model and metric: coverage, efficiency (lower is better), stability (lower std. dev.), and overall score (weighted aggregate score (0.4 coverage, 0.4 efficiency, 0.2 stability)). *All Features* dominates overall performance in three out of five models, while *Ext+Multi-CS* and *Multi-CS* excel in solver efficiency and large-scale coverage, respectively. See Appendix for full results.

CKB	Best Coverage	Best Efficiency	Best Stability	Best Overall
Arcade	All Features	Ext+Multi-CS	Full Pop	All Features
FQA	Ext+Multi-CS	All Features	Baseline	All Features
B2C	Ext+Multi-CS	Ext+Multi-CS	Full Pop	All Features
BusyBox	Ext+Multi-CS	Ext+Multi-CS	Weighted Basic	Extinction
EA	Multi-CS	Multi-CS	Extinction	Full Pop

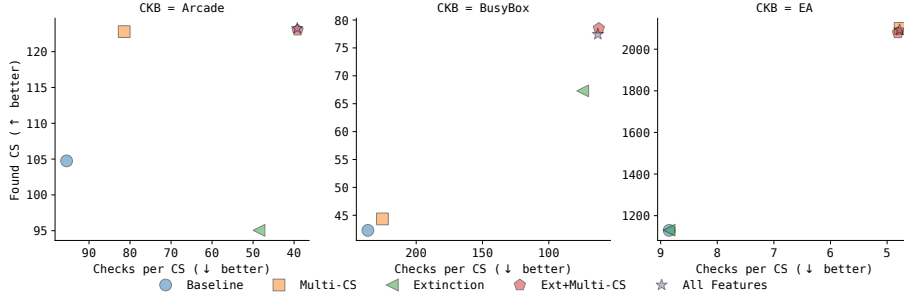


Figure 2.2: Trade-off between conflict set coverage (Found CS) and solver effort (Checks/CS) for three representative configuration knowledge bases. Configurations closer to the top-right indicate better coverage and lower solver cost. *All Features* and *Ext+Multi-CS* consistently lie near the Pareto frontier, while the *Baseline* lags behind in all settings. Full results for all models are shown in Figure 1 of the Appendix.

To ensure a comprehensive evaluation, we compare a baseline GA with eight enhanced configurations, each incorporating different combinations of four mechanisms (see Table 2.2): *multi-conflict extraction*, *full population reproduction*, *weighted genetic operators*, and *adaptive population extinction*. Each configuration is executed 20 times, and evaluated along three metrics: the average number of discovered conflict sets (*coverage*), the average number of consistency checks per conflict (*efficiency*), and the standard deviation of coverage (*stability*). These metrics are normalized per model using min-max scaling and aggregated into an overall score via the formula: $Overall = 0.4 \times NormCoverage + 0.4 \times NormEfficiency + 0.2 \times NormStability$. This aggregation favors configurations that balance performance and robustness, ensuring that unstable but high-coverage runs do not dominate the ranking.

All experiments were conducted on an Apple M1 Pro (8 cores, 16GB RAM) using the CHOCO solver for consistency checks (Prud’homme and Fages 2022). The population size was fixed to $N = 100$, with no assignment rate $P_0 = 0.7$, mutation rate $P_m = 0.1$, and a maximum of 100 generations. For *Multi-CS*, the conflict count was capped at 10 per individual. Extinction was triggered after 3 stagnant generations, with a maximum of 10 extinction cycles. Results across diverse knowledge bases indicate that performance remains stable under fixed parameter settings, suggesting robustness without the need for per-model tuning.

We first examine results on three representative CKBs. On the Arcade model (the leftmost subfigure in Figure 2.2), the *All Features* configuration delivers the strongest overall performance, achieving the highest coverage and a 58.9% reduction in consistency checks relative to the baseline. *Ext+Multi-CS* yields the best solver efficiency (39.15 checks per conflict), while *Full Pop* achieves the lowest variance (Table 2.3). Convergence analysis in Figure 2.3a shows that *All Features* rapidly accumulates conflict sets in early generations, whereas the baseline configuration stagnates early. This highlights the benefit of diversity preservation and adaptive resets in avoiding premature convergence.

In the BusyBox model (the middle subfigure in Figure 2.2), which is highly

2.5. Empirical Evaluation

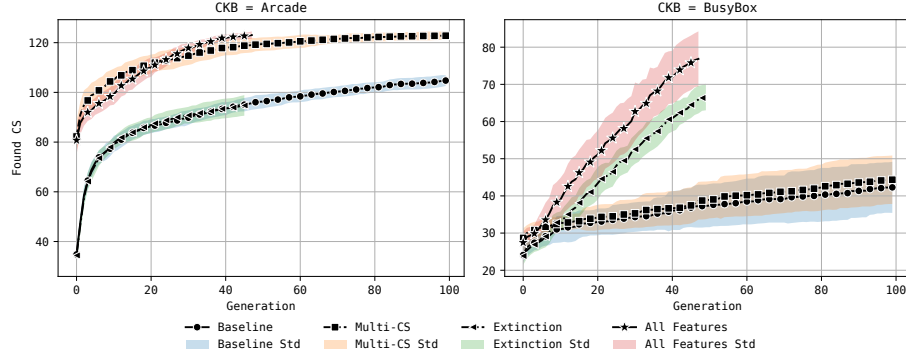


Figure 2.3: Convergence of conflict set discovery over generations for Arcade and BusyBox models. Solid lines denote the mean number of discovered conflict sets across 20 runs; shaded areas represent standard deviation. *All Features* and *Extinction* achieve faster and more stable convergence, particularly in the early generations. *Baseline* stagnates early, while *Multi-CS* improves total coverage but converges more slowly. Full plots are available in Appendix Figure 2.

constrained and solver-intensive, *Ext+Multi-CS* again outperforms all other configurations in both coverage and efficiency, recovering over 85% more conflicts while requiring 73% fewer consistency checks than the baseline. *Extinction* alone already provides significant gains, especially in early stages (Figure 2.3b), confirming its effectiveness in dense problem spaces. The *Weighted Basic* configuration achieves the best stability here (Table 2.3), showing that simpler enhancements may still yield benefits under certain conditions.

The EA model (the rightmost subfigure in Figure 2.2) is the largest and sparsest of the benchmarks. Here, *Multi-CS* emerges as the best-performing configuration across all dimensions, with the highest conflict set coverage (2,096.5 on average) and the lowest solver effort (4.77 checks per conflict). While *All Features* achieves similar results, *Full Pop* stands out as the most stable configuration, earning the best overall score (Table 2.3). Notably, *Extinction* contributes little in this context, as stagnation is not a limiting factor in such expansive solution spaces.

For the remaining models, FQA and B2C, we observe consistent patterns (see Tables 2–3 and Figures 1b, 1d, 2b, 2c in Appendix). *Ext+Multi-CS* and *All Features* continue to perform strongly across all metrics, while *Full Pop* reliably enhances stability. In denser models such as B2C, *Extinction* significantly boosts efficiency by accelerating early-stage discovery, though its benefit diminishes in models with less pronounced convergence bottlenecks.

Overall, our results demonstrate that each enhancement contributes complementary benefits. *Multi-CS* consistently improves coverage across all benchmarks. *Extinction* effectively reduces solver effort, particularly in dense or stagnation-prone models. *Full Pop* and *Weighted* configurations enhance stability and robustness, especially when initial population diversity is limited. Combined configurations, such as *All Features* and *Ext+Multi-CS*, offer the best trade-offs across metrics and knowledge base characteristics, supporting their applicability as default strategies for robust and scalable *lazy conflict detection*.

2.6 Conclusions

We presented a robust enhancement to *lazy conflict detection* by integrating multi-conflict extraction with genetic diversity control. Our method systematically addresses key limitations of the baseline approach, namely coverage gaps, premature convergence, and result instability, through four targeted mechanisms: *multi-conflict extraction*, *full population reproduction*, *weighted genetic operators*, and *adaptive extinction*. Comprehensive experiments on five real-world configuration knowledge bases demonstrate that our enhanced strategy significantly improves conflict coverage, reduces solver effort, and achieves high result stability. In particular, the *All Features* configuration consistently ranks among the best-performing variants across all evaluation metrics. Further trade-off and convergence analyses further illustrate the scalability and adaptability of our enhancements. Future work will explore adaptive orchestration of enhancement features, i.e., automatically activating mechanisms based on search dynamics, as well as automated parameter tuning to better tailor the approach to different application domains.

Chapter 3

INFORMEDQX: Informed Conflict Detection for Over-Constrained Problems

This chapter is based on the results documented in (Le, Felfernig, Tran, and Uta 2024). The author of this thesis led the algorithm design, carried out the literature research, implementation, empirical evaluation, and data analysis, and wrote major parts of the paper.

Conflict detection is relevant in various application scenarios, ranging from interactive decision-making to the diagnosis of faulty knowledge bases. Conflicts can be regarded as sets of constraints that cause an inconsistency. In many scenarios (e.g., constraint-based configuration), conflicts are repeatedly determined for the same or similar sets of constraints. This misses out on the valuable opportunity for leveraging knowledge reuse and related potential performance improvements, which are extremely important, specifically interactive constraint-based applications. In this chapter, we show how to integrate knowledge reuse concepts into non-instructive conflict detection.¹ We introduce the INFORMEDQX algorithm, which is a reuse-aware variant of QUICKXPLAIN. The results of a related performance analysis with the *Linux-2.6.33.3* configuration knowledge base show significant improvements in terms of runtime performance compared to QUICKXPLAIN.

3.1 Introduction

Conflict detection has many applications in constraint-based systems (and beyond) (Junker 2004; Rossi, Beek, and Walsh 2006). Examples thereof are *recommender systems* (Felfernig and Burke 2008), *knowledge-based configuration* (Junker 2006), *scheduling* (Baptiste et al. 2006), and *knowledge base testing and debugging* (Felfernig, Friedrich, Jannach, et al. 2004). In such scenarios, the task of conflict detection is to identify *minimal sets of constraints* (so-called *conflict sets* or *conflicts*) that can be interpreted as an explanation for the given inconsistency. Often associated with conflict detection is conflict resolution (often

¹The work presented in this chapter has been developed within the scope of the PARXCEL project, which is funded by the Austrian Research Promotion Agency (880657).

denotes as *diagnoses*), which focuses on resolving all identified conflicts so that knowledge base consistency can be restored (Reiter 1987; de Kleer and Williams 1987).

Especially in interactive settings, there is often a need to identify *preferred conflicts* (Junker 2004; O’Sullivan et al. 2007; Walsh 2007; Rossi, Venable, and Walsh 2011), i.e., conflicts whose resolution could be regarded as acceptable for a user. For example, users of a car configurator with strong preferences regarding an upper price limit are more inclined (in the case that a configurator cannot find a solution) to accept some relaxations of technical car features before accepting to further extend the pre-defined price limit.

With an increasing size and complexity of the underlying knowledge bases, there is a need to further improve the performance of the reasoning engines as well as related algorithms for conflict detection and resolution (Jannach, Schmitz, and Shchekotykhin 2015). To tackle related scalability issues, different approaches have been developed on-top of constraint solvers. Examples thereof are the application of machine learning for improving constraint solver performance (Popescu et al. 2022), algorithmic parallelization (Jannach, Schmitz, and Shchekotykhin 2015; Gent et al. 2018; Vidal et al. 2021; Le, Vidal Silva, et al. 2023), and knowledge compression techniques (Cheng and Yap 2005).

On the basis of a simplified working example from the domain of smartwatch configuration, we introduce a hybrid approach to conflict detection. The related algorithm (INFORMEDQX) that helps to decide, based on the *existing historic conflict data*, if an activation of conflict detection is still needed or if a preferred conflict has already been determined in previous configuration sessions. In this context, we focus on scenarios where the complete set of conflicts cannot be determined ahead due to complexity reasons, i.e., we want to reuse pre-existing conflicts and to figure out on the algorithmic level when the activation of a conflict detection algorithm is still needed due to the fact that further (more preferred) conflicts exist in a given constraint set.

Conflict detection algorithms have been proposed in the context of different knowledge representations such as constraint solving (Junker 2004; Shchekotykhin, Jannach, and Schmitz 2015) and SAT solving (Liffiton and Sakallah 2008; Marques-Silva and Previti 2014). In this context, the terms *minimal unsatisfiable cores* (MUC) or *minimal unsatisfiable subsets* (MUS) are often used as synonyms for minimal conflict sets. In this chapter, we focus on *complete* and *optimal* conflict detection algorithms, i.e., algorithms that are able to find a minimal conflict if one exists and at the same time find the optimal solution with regard to a predefined optimality criteria.

For demonstration purposes, we show how to exploit knowledge about existing conflicts in the context of the QUICKXPLAIN algorithm (Junker 2004) which is a divide-and-conquer based algorithm for the determination of preferred conflicts, i.e., conflicts that take into account pre-defined preferences of users. QUICKXPLAIN is knowledge-representation agnostic, i.e., applicable to different knowledge representations such as constraint solving (Junker 2004), SAT solving (Marques-Silva and Previti 2014), answer set programming (ASP) (Erdem, Gelfond, and Leone 2016), and description logics (DL) (McGuinness 2007), and does not exploit properties of a specific application domain.

Following a divide-and-conquer search regime, the underlying idea of QUICKXPLAIN is to divide the conflict solution space as efficiently as possible with the goal to identify subset-minimal conflict sets (one preferred minimal conflict

at a time). QUICKXPLAIN has shown to work efficiently in many application scenarios (Rodler 2022), however, with an increasing size and complexity of the underlying knowledge bases, further mechanisms are needed to assure algorithm scalability (Vidal et al. 2021).

In this chapter, we introduce a new algorithm (INFORMEDQX) that decides on a meta-level in which contexts there is a need to activate QUICKXPLAIN or whether a preferred conflict has already been determined in a previous configuration session. Interestingly, there exist scenarios in-between the two extreme cases that motivated the work presented in this chapter. We introduce intelligent conflict reuse, which is specifically needed in scenarios without complete conflict knowledge, i.e., where not all conflicts of a knowledge base are known ahead – predetermining such conflicts in the general case is known to be a hard problem (Gregoire, Mazure, and Piette 2008). In such situations, we have to find a solution for exploiting incomplete conflict knowledge, which helps to improve the performance of conflict detection as a whole.

The major contributions of this chapter are the following:

1. We introduce an algorithm (INFORMEDQX) and corresponding principles that help to make conflict search more efficient given a setting where some parts of the conflict space are already known.
2. On the basis of an evaluation with a real-world configuration knowledge base, we show a significantly improved conflict detection performance compared to the basic QUICKXPLAIN algorithm.
3. INFORMEDQX is not limited to the application in constraint solving scenarios but can as well be applied in the context of other knowledge representations such as ASP, SAT, and DL.

3.2 Example Configuration Setting

In the context of a simplified example from the domain of *constraint-based smart watch configuration*, we now introduce the definitions of a *configuration task* (Definition 3.1) and a corresponding *configuration* (Definition 3.2) (see also (Felfernig, Hotz, et al. 2014)). The following discussions are based on a constraint satisfaction problem (CSP) knowledge representation (Rossi, Beek, and Walsh 2006).

Definition 3.1 (Configuration task and Knowledge base). A configuration task (V, C) can be defined as a CSP where (1) $V = \{v_1, v_2 \dots v_n\}$ is a set of variables with finite domain definitions $dom(v_i)$ indicating the domain of each variable $v_i \in V$ and (2) $C = C_{KB} \cup C_R$ is a set of constraints restricting possible solutions of a configuration task. In this context, C_{KB} represents a set of domain-specific constraints and C_R represents a set of user requirements. Finally, (V, C_{KB}) is denoted as configuration knowledge base.

Given such a definition of a configuration task, we now introduce the definition of a corresponding configuration.

Definition 3.2 (Configuration). A configuration (solution) S for a given configuration task (V, C) is an *assignment* $A = \{v_1 = a_1 \dots v_n = a_n\}$, $a_i \in dom(v_i)$.

3.2. Example Configuration Setting

Table 3.1: Configuration task constraints $C_{KB} = \{c_0 \dots c_{10}\}$ and $C_R = \{c_{11} \dots c_{19}\}$. $c_0 : \text{Smartwatch} = t$ is a *root constraint*, avoiding the derivation of empty configurations. $c_7 : \text{Screen} \leftrightarrow \text{xor}(\text{Analog}, \text{High Resolution}, \text{E-ink})$ returns *true*, if exactly one out the three variables is *true*, otherwise it returns *false*.

CSP Representation	
Domain-specific Constraints (C_{KB})	
c_0	$\text{Smartwatch} = t$
c_1	$\text{Smartwatch} \leftrightarrow \text{Connector}$
c_2	$\text{Smartwatch} \leftrightarrow \text{Screen}$
c_3	$\text{Camera} \rightarrow \text{Smartwatch}$
c_4	$\text{Compass} \rightarrow \text{Smartwatch}$
c_5	$\text{Speaker} \rightarrow \text{Smartwatch}$
c_6	$\text{Connector} \leftrightarrow (\text{GPS} \vee \text{Cellular} \vee \text{Wifi} \vee \text{Bluetooth})$
c_7	$\text{Screen} \leftrightarrow \text{xor}(\text{Analog}, \text{High Resolution}, \text{E-ink})$
c_8	$\text{Camera} \rightarrow \text{High Resolution}$
c_9	$\text{Compass} \rightarrow \text{GPS}$
c_{10}	$\neg(\text{Cellular} \wedge \text{Analog})$
User Requirements (C_R)	
c_{11}	$\text{Cellular} = t$
c_{12}	$\text{Bluetooth} = f$
c_{13}	$\text{Analog} = t$
c_{14}	$\text{Compass} = t$
c_{15}	$\text{Wifi} = t$
c_{16}	$\text{GPS} = f$
c_{17}	$\text{Speaker} = t$
c_{18}	$\text{Camera} = t$
c_{19}	$\text{E-ink} = f$

S is *valid* if it is *complete* (each variable in V has a value) and *consistent* (S fulfills the constraints in C).

Based on these definitions, the definition of a simplified smart watch configuration task looks like as follows (see Example 3.1). A configurable *Smartwatch* must have at least one type of *Connector* and a *Screen*. The *Connector* can be of one or more of the types of *GPS*, *Cellular*, *Wifi*, and *Bluetooth*. The *Screen* can be either *Analog*, *High Resolution*, or *E-ink*. Besides, a *Smartwatch* may include a *Camera*, a *Compass*, and a *Speaker*. Furthermore, *Compass* requires a *GPS*, *Camera* requires a *High Resolution*, and *Cellular* and *Analog* exclude each other. A constraint-based representation of these restrictions can be found in Table 3.1.

Example 3.1 (*Smartwatch* configuration task). A CSP-based *Smartwatch* configuration task (V, C) is the following :

- $V = \{\text{Smartwatch}, \text{Connector}, \text{Screen}, \text{Camera}, \text{Compass}, \text{Speaker}, \text{GPS}, \text{Cellular}, \text{Wifi}, \text{Bluetooth}, \text{Analog}, \text{High Resolution}, \text{E-ink}\}$
- $\text{dom}(\text{Smartwatch}) = \{(t)\text{true}, (f)\text{false}\} \dots \text{dom}(\text{E-ink}) = \{(t)\text{true}, (f)\text{false}\}$
- $C_{KB} = \{c_0 \dots c_{10}\}$, $C_R = \{c_{11} \dots c_{19}\}$.

In our example, some of the user requirements C_R (see Table 3.1) are inconsistent with the constraints in C_{KB} , i.e., the solver is not able to find a related

configuration. For instance, c_{11} and c_{13} in C_R are inconsistent with c_{10} in C_{KB} . Therefore, no solution can be found for this configuration task. In such situations, we are interested in explanations as to why no solution could be identified. Minimal conflict sets, also denoted as minimal unsatisfiable subsets, are a means often used to explain such inconsistent situations. In the following, we formally introduce the notion of a *minimal conflict set* and also introduce related preference criteria, i.e., criteria regarding the degree of preferredness of individual conflict sets. In this context, we also discuss the major properties of QUICKX-PLAIN (Junker 2004) which is used for demonstration purposes throughout this chapter.

3.3 Determining Preferred Minimal Conflicts

Conflict sets are constraint sets that are responsible for an inconsistency, i.e., a situation in which no solution can be found. With $\text{consistent}(C)$, we express that a constraint set C is consistent, and $\text{inconsistent}(C)$ reflects situations where no solution can be found for C . Definition 3.3 introduces the concept of conflict set minimality on the basis of subset minimality (i.e., not minimal cardinality): if CS is a minimal conflict, no proper subset of C can be a minimal conflict.

Definition 3.3 (Conflict set). A *conflict set* is a set $CS \subseteq C_R : \text{inconsistent}(CS \cup C_{KB})$. CS is *minimal* iff $\nexists CS' \subset CS : \text{inconsistent}(CS' \cup C_{KB})$.

An example of minimal conflict sets in the context of our example configuration task is the following (see Example 3.2).

Example 3.2 (Minimal conflict sets). Given the configuration task $(V, C = C_{KB} \cup C_R)$ presented in Example 3.1, we are able to identify the following minimal conflict sets: $CS_1 = \{c_{11}, c_{13}\}$, $CS_2 = \{c_{13}, c_{18}\}$, and $CS_3 = \{c_{14}, c_{16}\}$. The minimality property is fulfilled since $\nexists CS_4 : CS_4 \subset CS_1$, $\nexists CS_5 : CS_5 \subset CS_2$, and $\nexists CS_6 : CS_6 \subset CS_3$.

Preferred Minimal Conflict. To resolve inconsistencies in *interactive settings* such as configuration (O’Sullivan et al. 2007; Felfernig, Hotz, et al. 2014), a user has to resolve conflicts consisting of constraints that represent user requirements ($c_i \in C_R$). In this context, a constraint that is of low importance for the user is a preferred candidate for being part of a conflict set.

To make the preference degree of a conflict set more transparent, we introduce the following definitions of a *strict total order* (Definition 3.4) and a corresponding *preferred conflict* (Definition 3.5). These definitions provide a way to clearly identify a preferred conflict set among a set of candidates (see also (Junker 2004; Marques-Silva and Previti 2014)).

Definition 3.4 (Strict total order). A *strict total order* $<$ over the constraints in $C_R = \{c_1 \dots c_m\}$ is represented as $\langle c_1 < c_2 < \dots < c_m \rangle$ where $\forall (c_i, c_{i+1}) : c_i$ is preferred over c_{i+1} .

On the basis of such an ordering of individual constraints part of a conflict set, we are able to characterize the preference degree of a conflict set on the basis of a pairwise comparison (see (Junker 2004)). Given a strict total order $<$ of a set of constraints, there exists a unique preferred conflict set (Junker 2004).

3.3. Determining Preferred Minimal Conflicts

Definition 3.5 (Preferred conflict). Given a *strict total order* $<$ over C_R , a set $X \subseteq C_R$ is preferred over another set $Y \subseteq C_R$ (denoted $X >_{lex} Y$) iff $\exists_{i \leq k \leq m} : c_k \in Y \setminus X$ and $X \cap \{c_1 \dots c_{k-1}\} = Y \cap \{c_1 \dots c_{k-1}\}$. A minimal conflict set CS is a (lexicographically) preferred conflict iff $\forall CS' \neq CS : CS >_{lex} CS'$.

Following Definition 3.5, a preferred conflict in our example configuration task is the following (see Example 3.3).

Example 3.3 (Preferred conflict). Given the minimal conflict sets $CS_1 = \{c_{11}, c_{13}\}$, $CS_2 = \{c_{13}, c_{18}\}$, $CS_3 = \{c_{14}, c_{16}\}$, and a corresponding total ordering of $\langle c_{11} < c_{12} < \dots < c_{19} \rangle$, the preferred conflict (conflict set) can be determined as follows:

- CS_3 is preferred over CS_1 (denoted as $CS_3 >_{lex} CS_1$) since $\exists c_{11} \in CS_1 \setminus CS_3$ with $CS_1 \cap \emptyset = CS_3 \cap \emptyset$.
- CS_3 is preferred over CS_2 (denoted as $CS_3 >_{lex} CS_2$) since $\exists c_{13} \in CS_2 \setminus CS_3$ with $CS_2 \cap \{c_{11}, c_{12}\} = CS_3 \cap \{c_{11}, c_{12}\}$.
- The preferred conflict set is CS_3 .

In Example 3.3, the preferred conflict set is CS_3 . Following the transitive properties of such lexicographical orderings (Brewka 1989; Junker 2004), no further comparisons are needed for CS_1 and CS_2 .

Preference formula. Following Definition 3.5, we now introduce a numerical evaluation function (based on bitmap indexing (O’Neil 1987)) which is used in the following to numerically evaluate conflict preference (see Formulae 3.1–3.3).

$$preference(CS) = MAX - \sum_{c_i \in CS} importance(c_i) \quad (3.1)$$

$$importance(c_i) = 2^{|C_R| - i} \quad (3.2)$$

$$MAX = 2^{|C_R|} - 1 \quad (3.3)$$

The preference of a conflict set CS ($preference(CS)$) can be determined by evaluating the sum of the individual user preferences regarding constraints in CS (see Definition 3.4). In this context, we apply a so-called bitmap indexing (O’Neil 1987) following the idea of evaluating individual constraint rankings following a binary system ($2^{|C_R| - 1}, 2^{|C_R| - 2} \dots 1$), where $|C_R|$ denotes the C_R ’s cardinality. In our working example, $preference(CS_1 = \{c_{11}, c_{13}\}) = 191$ assuming $importance(c_{11}) = 256$ and $importance(c_{13}) = 64$. A complete evaluation of the conflict sets of our working example is depicted in Table 3.2.

The higher the sum over the importance values of constraints in CS , the lower the probability that CS is the preferred conflict set. In this context, MAX (see Formula 3.3) refers to the least preferred conflict set including all elements of C_R with a related theoretical preference value of 0. Consequently, the lower the total importance of constraints in CS , the higher the preference for CS . The theoretical upper bound of $preference(CS)$ is $MAX - 1$ assuming that the CS with the lowest importance (i.e., 1) represents a singleton conflict set. Finally, i ($i \in \{1 \dots |C_R|\}$) denotes the i^{th} constraint in CS , for example, the index of i of c_{11} in our example set of requirements C_R is 1.

Table 3.2: Preference values for the conflict sets CS_1 , CS_2 , and CS_3 given $\langle c_{11} < c_{12} < \dots < c_{19} \rangle$ as the strict total ordering of the user requirements in C_R . In this setting, CS_3 is regarded as preferred minimal conflict set.

C_R	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	c_{19}	$preference(CS_i)$
i	1	2	3	4	5	6	7	8	9	
$importance(c_i)$	256	128	64	32	16	8	4	2	1	
CS_1	$\times$	—	$\times$	—	—	—	—	—	—	$511 - (256 + 64) = 191$
CS_2	—	—	$\times$	—	—	—	—	$\times$	—	$511 - (64 + 2) = 445$
CS_3	—	—	—	$\times$	—	$\times$	—	—	—	$511 - (32 + 8) = 471$

3.3. Determining Preferred Minimal Conflicts

Algorithm 3.1 QUICKXPLAIN(C_R, C_{KB}) : CS

```

1: if CONSISTENT( $C_{KB} \cup C_R$ ) then
2:   print ‘no conflict’
3:   return  $\emptyset$ 
4: else if  $C_R = \emptyset$  then
5:   print ‘conflict detection not possible’
6:   return  $\emptyset$ 
7: else
8:   return QX( $\emptyset, C_R, C_{KB}$ )
9: end if

```

Algorithm 3.2 QX($\Delta, C_R = \{c_1 \dots c_n\}, C_{KB}$) : CS

```

1: if  $\Delta \neq \emptyset$  and INCONSISTENT( $C_{KB}$ ) then
2:   return  $\emptyset$ 
3: end if
4: if  $|C_R| = 1$  then
5:   return  $C_R$ 
6: end if
7:  $k = \lfloor \frac{n}{2} \rfloor$ 
8:  $C_{Ra} \leftarrow c_1 \dots c_k; C_{Rb} \leftarrow c_{k+1} \dots c_n$ 
9:  $\Delta_2 \leftarrow$  QX( $C_{Rb}, C_{Ra}, C_{KB} \cup C_{Rb}$ )
10:  $\Delta_1 \leftarrow$  QX( $\Delta_2, C_{Rb}, C_{KB} \cup \Delta_2$ )
11: return  $\Delta_1 \cup \Delta_2$ 

```

QUICKXPLAIN. QUICKXPLAIN (Junker 2004) (see Algorithms 3.1 and 3.2) is an algorithm that can be applied to determine *preferred minimal conflict sets* (one minimal conflict set at a time) in a given set of constraints (in our case, C_R). In this context, C_R is denoted as *consideration set* whereas the second parameter represents the so-called *background knowledge*, i.e., a set of constraints that is considered consistent – in our case, the background knowledge is represented by the set of domain-specific constraints C_{KB} . QUICKXPLAIN divides set C_R into two subsets (C_{Ra} and C_{Rb}). If one subset (e.g., C_{Ra}) is inconsistent, then conflict detection should be applied to this subset. The other subset (C_{Rb}) must not be further analyzed since at least one conflict exists in C_{Ra} . This way, the consideration set C_R can be reduced by half with a single consistency check. The constraint ordering in C_R conforms to the definition of the strict total ordering (see Definition 3.4). QUICKXPLAIN is activated if the background knowledge C_{KB} is *inconsistent* with C_R (see line 8 of Algorithm 3.1).

The core algorithm is implemented in the function QX (Algorithm 3.2) that determines a *minimal conflict set* in a divide-and-conquer fashion. An execution trace of QUICKXPLAIN on the basis of our working example is depicted in Figure 3.1. The algorithm QX adds additional constraints (from C_{Rb}) to C_{KB} as long as the resulting constraint set remains consistent. If it is inconsistent, then the algorithm leaves out the remaining constraints. For example, in the activation [3], the set $C_{KB} \cup \{c_{14} \dots c_{19}\}$ is inconsistent and thus, the remaining constraints ($\{c_{11} \dots c_{13}\}$) can be removed.

On the other hand, if the background knowledge is consistent and only one constraint remains that induces the inconsistency, this constraint must be part

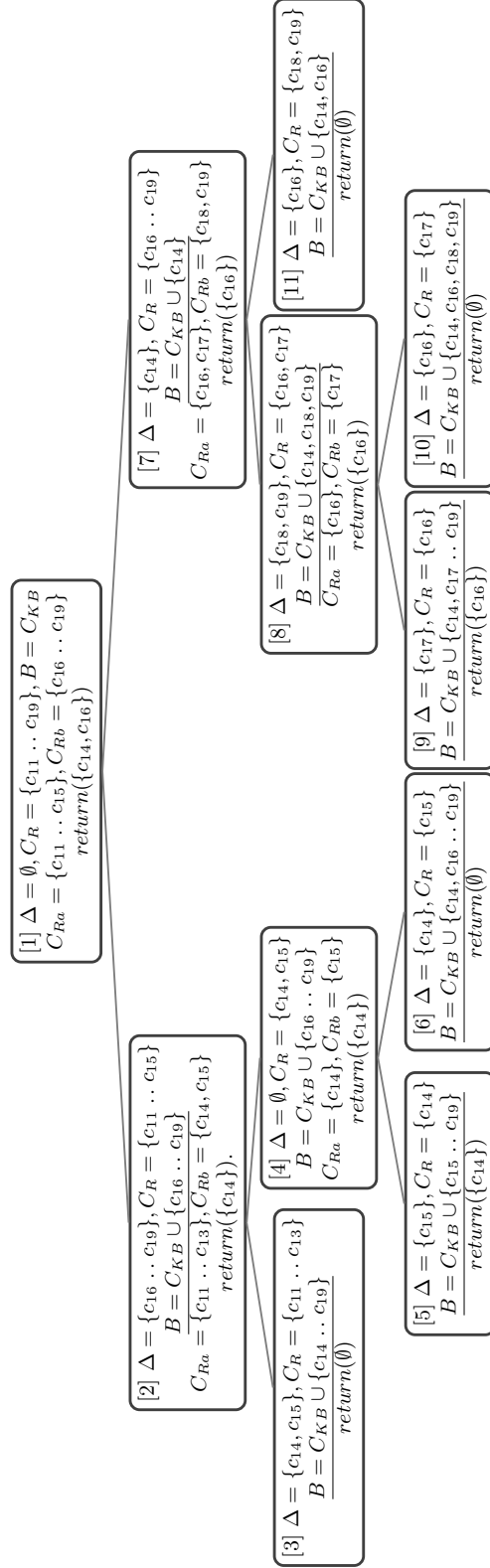


Figure 3.1: QX execution trace for $C_R = \{c_{11} \dots c_{19}\}$ and $B = C_{KB}$. Underlined B s denote QX consistency checks. For example, in the activation [2] of the QX function, the consistency check activated is $B = C_{KB} \cup \{c_{16} \dots c_{19}\}$.

of the conflict set. For example, in the activation [9], the background knowledge consists of the constraints $C_{KB} \cup \{c_{14}, c_{17} \dots c_{19}\}$ and c_{16} remains as a single constraint. It is clear that c_{16} is part of the conflict set since $C_{KB} \cup \{c_{14}, c_{17} \dots c_{19}\}$ is consistent but $C_{KB} \cup \{c_{14}, c_{17} \dots c_{19}\} \cup \{c_{16}\}$ is inconsistent.

3.4 INFORMEDQX

General idea. QUICKXPLAIN works efficiently in many scenarios, however, it shows performance issues in the context of large and complex knowledge bases (Vidal et al. 2021). To tackle this challenge, we introduce INFORMEDQX (as a QUICKXPLAIN variant) which exploits known conflicts (e.g., from previous configuration sessions) to efficiently narrow down the conflict analysis space.

This exploitation (conflict reuse) can take place in two basic settings. First, a direct reuse (without activating QX) is possible if a specific predetermined conflict is the preferred conflict of the user requirements C_R . Second, when some predetermined conflicts match with the constraints of the user requirements C_R but its preferred conflict is not available yet, the *currently* preferred conflict of the predetermined conflicts should be identified. In this setting, we can reduce the size of C_R by pruning certain constraints that would not fit well with the more preferred conflict (the conflict with a higher preference value as described in Formula 3.1). This pruned set, C_P , can then be passed to QX, making the process of finding conflicts faster.

We assume that N denotes the set of predetermined minimal conflicts. It is not necessarily complete (i.e., not all conflicts of C_R are in N), only those that have already been identified. The determination of the preferred conflict based on N can be formally described in the following scenarios:

1. **Scenario 1** - If there are no conflicts in N or if none of the conflicts in N match the constraints of C_R , then the original QUICKXPLAIN (QX) algorithm must be used.
2. **Scenario 2** - If there are conflicts in N that belong to C_R , then the currently preferred conflict N_{cp} out of these conflicts can be determined. Additionally, we can check if there are any conflicts in C_R that are even more preferred than N_{cp} . If so, there are two possible sub-scenarios:
 - (a) **Scenario 2.1** - *There do not exist any more preferred conflicts*: We do not need to activate QX, i.e., the *currently* preferred conflict is the preferred conflict. For instance, given $C_R = \{c_{11} \dots c_{19}\}$, $N = \{N_1 = \{c_{11}, c_{13}\}, N_2 = \{c_{13}, c_{18}\}, N_3 = \{c_{14}, c_{16}\}\}$, $N_{cp} = \{c_{14}, c_{16}\}$ (see Example 3.3), and no conflicts in C_R that are *more* preferred over N_{cp} , N_{cp} then becomes the preferred conflict of C_R and the activation of QX is not needed (see Figure 3.2).
 - (b) **Scenario 2.2** - *There exists at least one more preferred conflict*: We activate QX with the pruned set C_P . Based on Definitions 3.4 and 3.5, the constraints of C_R with a lower lexicographical order than the *currently* preferred conflict can be omitted. For instance, given $C_R = \{c_{11} \dots c_{19}\}$, and $N = \{N_1 = \{c_{11}, c_{13}\}, N_2 = \{c_{13}, c_{18}\}\}$, the *currently* preferred conflict set is $N_{cp} = \{c_{13}, c_{18}\}$, the pruned set of C_R should be $C_P = \{c_{13} \dots c_{19}\}$. The reason is that the constraints

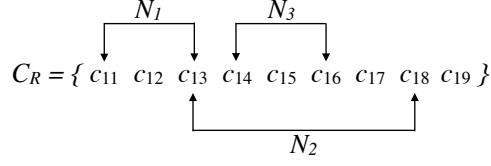


Figure 3.2: An illustration of the *Scenario 2.1* with $C_R = \{c_{11} \dots c_{19}\}$, and $N = \{N_1 = \{c_{11}, c_{13}\}, N_2 = \{c_{13}, c_{18}\}, N_3 = \{c_{14}, c_{16}\}\}$. In this case, N_3 will be the *currently* preferred conflict N_{cp} , as well as the preferred conflict of C_R .

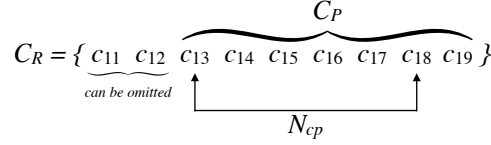


Figure 3.3: An illustration of the *Scenario 2.2* with $C_R = \{c_{11} \dots c_{19}\}$, and $N = \{N_1 = \{c_{11}, c_{13}\}, N_2 = \{c_{13}, c_{18}\}\}$. In this case, $N_{cp} = \{c_{13}, c_{18}\}$ and $C_P = \{c_{13} \dots c_{19}\}$.

$\{c_{11}, c_{12}\}$ cannot be part of the *more* preferred conflict and therefore are omitted. Consequently, QX is activated with C_P , whose size is much smaller than this of the original set C_R (see Figure 3.3).

Construction of N . Identifying a N_{cp} out of N is an expensive computational task. To tackle this issue, N is constructed on the basis of the Binary Decision Diagram (BDD) (Cheng and Yap 2005). In this context, the approach requires an offline process where a BDD is built to compress all identified conflict sets.

INFORMEDQX algorithm. The INFORMEDQX algorithm (see Algorithm 3.3) comes into play when the existing background knowledge C_{KB} is *inconsistent* with C_R . The initial step of Algorithm 3.3 identifies the *currently* preferred conflict N_{cp} from the set N (*line 1*). The FINDPREFERREDCONFLICT *function* determines this preferred conflict based on Definition 3.5 and Formula 3.1. Initially, this function extracts from N all the conflict sets associated with C_R utilizing the *findAll* operation of the BDD diagram (Cheng and Yap 2005). It then calculates the preferences for each of these identified conflict sets. Eventually, the function returns the conflict set with the highest preference value. Moreover, if N is empty, the conflict extraction process from N is ignored, and the function returns an empty set. Additionally, if no conflicts of C_R are present in N , the function returns an empty set.

If $N_{cp} = \emptyset$, i.e., N is empty or no conflicts of C_R stay in N (*line 2*), then the *traditional* QUICKXPLAIN is activated for C_R (*line 3*). Otherwise, in line with *Scenario 2.2*, certain unnecessary constraints in C_R , which will not be part of the *more* preferred conflict than the *currently* preferred conflict N_{cp} , are omitted (*line 5*). After pruning C_R , the algorithm examines whether there exist further preferred conflicts in the pruned set C_P (*lines 6–14*). This is addressed by evaluating if the inconsistency of C_P with C_{KB} remains unchanged even when removing a constraint $c \in N_{cp}$ from C_P (*lines 9–10*). In our approach, for each

Algorithm 3.3 INFORMEDQX(C_R, C_{KB}, N) : CS

```

1:  $N_{cp} \leftarrow \text{FINDPREFERREDCONFLICT}(N, C_R)$ 
2: if  $N_{cp} = \emptyset$  then
3:   return QX( $\emptyset, C_R, C_{KB}$ )
4: else
5:    $C_P \leftarrow \text{PRUNE}(C_R, N_{cp})$ 
     {begin - examination for a further preferred conflict}
6:    $prev\_c \leftarrow \emptyset$ 
7:   for all  $c \in N_{cp}$  do
8:      $idx\_c \leftarrow \text{INDEX}(c, C_P)$ 
9:      $C'_P \leftarrow prev\_c \cup (C_P \setminus \{C_P[i] : \forall i \in [0 \dots idx\_c]\})$ 
10:    if INCONSISTENT( $C_{KB} \cup C'_P$ ) then
11:      return QX( $\emptyset, C'_P, C_{KB}$ )
12:    end if
13:     $prev\_c \leftarrow prev\_c \cup c$ 
14:  end for
     {end - examination for a further preferred conflict}
15:  return  $N_{cp}$ 
16: end if

```

Table 3.3: An illustration of *lines 6–14* in Algorithm 3.3 for checking whether there exists further conflicts in $C_P = \{c_{14} \dots c_{19}\}$, where $N_{cp} = \{c_{14}, c_{16}\}$.

<i>loop</i>	$c \in N_{cp}$	C'_P	INCONSISTENT($C_{KB} \cup C'_P$)
1	c_{14}	$\{c_{15} \dots c_{19}\}$	<i>false</i>
2	c_{16}	$\{c_{14}, c_{17} \dots c_{19}\}$	<i>false</i>

$c \in N_{cp}$, not only c itself is removed, but also constraints not present in N_{cp} yet confirmed as consistent with C_{KB} in the previous checks (*line 9*). For instance, since $C'_P = \{c_{15} \dots c_{19}\}$ is consistent with C_{KB} (see *loop 1* in Table 3.3), i.e., no conflict between c_{15} and other constraints in C'_P , c_{15} (besides c_{16}) is omitted from C_P in the next iteration (see *loop 2* in Table 3.3). Besides, c_{14} remains untouched because it belongs to $N_{cp} = \{c_{14}, c_{16}\}$.

During the examination, if a constraint $c \in N_{cp}$ satisfies the check (*line 10*), the algorithm triggers QX for the current constructed subset C'_P (*line 11*). Should the examination fail for all constraints of the conflict N_{cp} , the algorithm returns N_{cp} without activating QX (*line 15*). In other words, N_{cp} represents the preferred conflict of C_R . Figure 3.4 depicts the INFORMEDQX execution trace for our working example, highlighting the reduction in the number of required consistency checks to 6, in contrast to the 9 checks in QUICKXPLAIN (as shown in Figure 3.1).

3.5 Analysis of INFORMEDQX

We will now delve into a theoretical analysis of INFORMEDQX and proceed to evaluate its performance in comparison with QUICKXPLAIN (Junker 2004).

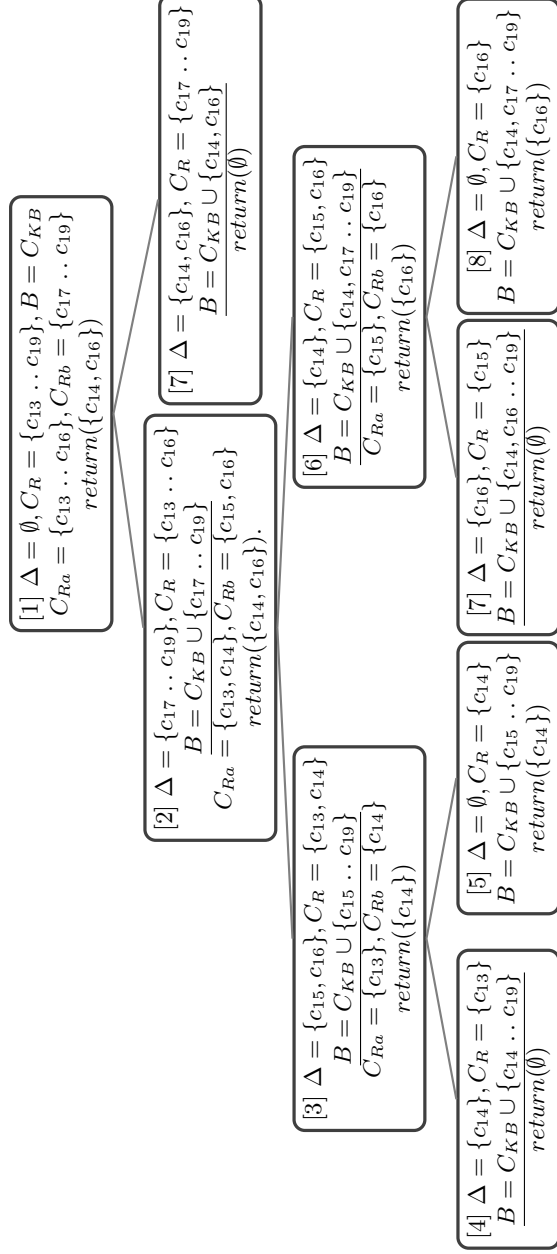


Figure 3.4: INFORMEDQX execution trace for $C_R = \{c_{11} \dots c_{19}\}$, $B = C_{KB}$, $N = \{\{c_{11}, c_{13}\}, \{c_{13}, c_{18}\}\}$. INFORMEDQX identifies the currently preferred conflict $N_{cp} = \{c_{13}, c_{18}\}$, and reduces C_R to $C_P = \{c_{13} \dots c_{19}\}$ (see Figure 3.3). The input of QX is $C_R = \{c_{13} \dots c_{19}\}$, $B = C_{KB}$.

3.5. Analysis of INFORMEDQX

Table 3.4: The *complexity* of INFORMEDQX and QUICKXPLAIN

method	best case	worst case
IQX	$m + 1$	$2c \times \log_2(\frac{n}{c}) + 2c + m$
QX	$\log_2(\frac{n}{c}) + 2c$	$2c \times \log_2(\frac{n}{c}) + 2c$

Table 3.5: *Number of solver calls / number of reused conflict sets / average run-time performance* (in seconds) of INFORMEDQX versus QUICKXPLAIN needed for determining the preferred conflict set after 20 iterations. $|CS|$ denotes the cardinality of the preferred conflict set. $N_{cp} = \emptyset$, $N_{cp} \neq \emptyset \wedge N_{cp} \neq CS$, $N_{cp} = CS$ indicate three following cases of N_{cp} returned by FINDPREFERREDCONFLICT: (1) there are no conflicts of C_R in N , (2) N_{cp} is not the preferred conflict CS , and (3) N_{cp} is the preferred conflict. Please note that, *column 2* shows zero reused conflict sets for QUICKXPLAIN since this algorithm does not utilize any reuse mechanisms.

$ CS $	QUICKXPLAIN	INFORMEDQX		
		$N_{cp} = \emptyset$	$N_{cp} \neq \emptyset \wedge N_{cp} \neq CS$	$N_{cp} = CS$
2	8 / 0 / 3.58	8 / 0 / 3.64	4 / 3 / 2.57	0 / 4 / 1.97
4	12 / 0 / 7.47	12 / 0 / 7.59	10 / 3 / 6.21	0 / 4 / 4.20
8	22 / 0 / 15.92	22 / 0 / 18.91	18 / 4 / 12.60	0 / 14 / 9.07

QUICKXPLAIN Complexity. The *worst-case* complexity of QX in terms of the number of needed consistency checks for determining one minimal unsatisfiable subset CS is $2c \times \log_2(\frac{n}{c}) + 2c$, where c is the *size of the minimal conflict set*, n is the *number of constraints* in C_R , and $2c$ represents the branching factor and the number of leaf-node consistency checks (Junker 2004). The *best-case* complexity is $\log_2(\frac{n}{c}) + 2c$. In the worst case, each faulty element is located in a different path of the search tree. The factor $\log_2(\frac{n}{c})$ represents the depth of a path of the QX search tree. Under the best circumstance, every constraint belonging to a conflict is included in a single path of the search tree.

INFORMEDQX Complexity. The complexity of INFORMEDQX can be calculated according to the following factors: (1) *the number of needed consistency checks* for determining one minimal unsatisfiable subset CS and (2) *the complexity of BDD queries* (m). In the *worst-case*, the complexity emerges as the cumulative sum of these two factors, which is $2c \times \log_2(\frac{n}{c}) + 2c + m$ where m is the number of the nodes in N 's BDD. In this scenario, the only distinction in the complexity of the algorithms arises from the extra factor m that is the complexity of BDD queries in the FINDPREFERREDCONFLICT. The corresponding *best-case* complexity is $m + 1$. In this scenario, since the preferred conflict of C_R is already known, the complexity of INFORMEDQX is the sum of the complexity of BDD queries in the FINDPREFERREDCONFLICT and one consistency check that confirms the preferred conflict.

INFORMEDQX Runtime Performance. We have evaluated the performance of INFORMEDQX compared to QUICKXPLAIN on the basis of the *Linux-*

2.6.33.3 configuration knowledge base taken from Diverso Lab’s benchmark² (Heradio et al. 2022). The characteristics of this knowledge base are the following: #features = 6,467; #relationships = 6,322; and #cross-tree constraints = 7,650. For this knowledge base, we used a genetic approach (Uran and Felfernig 2018) to synthesize and collect 136 minimal conflict sets, whose cardinality varies from 2 to 8.³

All experiments have been conducted with an Apple M1 Pro (8 cores) computer with 16-GB RAM. For evaluation purposes, we used the CHOCO solver⁴ to perform consistency checks and JAVABDD v6.0.0⁵ to build a BDD of identified conflict sets. Each entry in Table 3.5 represents *number of needed constraint solver calls (consistency checks) / number of reused conflict sets / average runtime (in seconds)* for both, INFORMEDQX and QUICKXPLAIN with a repetition of 20 per setting.

The results from the experiment displayed in Table 3.5 demonstrate that INFORMEDQX performs better than QUICKXPLAIN in all scenarios where known conflicts can be utilized. This improvement is most noticeable when the number of reused conflict sets is greater than zero. Although INFORMEDQX may have slightly inferior performance compared to QUICKXPLAIN when there are no conflicts of the users requirements C_R found in N (as seen in *column 3*), it significantly outperforms QUICKXPLAIN when conflict sets are reused (as seen in the two last cases of N_{cp} in *columns 4 and 5*). According to our experimental findings, BDD queries exhibit notably shorter average runtime (merely from 3 to 5 msec) compared to the total runtime of INFORMEDQX.

3.6 Conclusions

In this chapter, we have proposed an algorithm so-called INFORMEDQX as an improved version of the QUICKXPLAIN algorithm. The proposed algorithm resolves run-time performance issues in scenarios where the knowledge base is complex and exponentially large. With our algorithm, only conflicts that have been predetermined in previous conflict detection sessions are taken into account. This way, the algorithm helps to decrease the conflict analysis space and hence, speeds up the conflict detection process. The evaluation results show that INFORMEDQX outperforms QUICKXPLAIN in most of the evaluation cases.

Open topics for future research are the following: (1) performing more in-depth evaluations on the basis of other industrial configuration knowledge bases, and (2) applying the *informed* mechanisms in the context of diagnosis scenarios (e.g., integrating *informed mechanisms* into FASTDIAG (Felfernig, Schubert, and Zehentner 2012)).

²<https://github.com/diverso-lab/benchmarking>

³To ensure the reproducibility of the results, we used the seed value of 141982L for the random number generator.

⁴choco-solver.org

⁵<https://github.com/com-github-javabdd/com.github.javabdd>

Chapter 4

FASTDIAGP: An Algorithm for Parallelized Direct Diagnosis

This chapter is based on the results documented in (Le, Vidal Silva, et al. 2023). The author of this thesis designed the parallelization algorithm and carried out the literature research, implementation, empirical evaluation, and data analysis, and wrote major parts of the paper.

Constraint-based applications attempt to identify a solution that meets all defined user requirements. If the requirements are inconsistent with the underlying constraint set, algorithms that compute diagnoses for inconsistent constraints should be implemented to help users resolve the “no solution could be found” dilemma. FASTDIAG is a typical direct diagnosis algorithm that supports diagnosis calculation without predetermining conflicts. However, this approach faces runtime performance issues, especially when analyzing complex and large-scale knowledge bases. In this chapter, we propose a novel algorithm, so-called FASTDIAGP, which is based on the idea of speculative programming.¹ This algorithm extends FASTDIAG by integrating a parallelization mechanism that anticipates and pre-calculates consistency checks requested by FASTDIAG. This mechanism helps to provide consistency checks with fast answers and boosts the algorithm’s runtime performance. The performance improvements of our proposed algorithm have been shown through empirical results using the *Linux-2.6.33.3* configuration knowledge base.

4.1 Introduction

In many applications of constraint-based representations such as *knowledge-based configuration* (Stumptner 1997), *recommendation* (Felfernig and Burke 2008), *automated analysis of feature models* (Benavides, Segura, and Ruiz-Cortés 2010), and *scheduling* (Castillo et al. 2005), there exist some scenarios where over-constrained formulations occur in the underlying constraint sets (Felfernig, Schubert, Mandl, et al. 2010; Jannach, Schmitz, and Shchekotykhin

¹The work presented in this chapter has been partially funded by the PARXCEL project, supported by the Austrian Research Promotion Agency (880657), and by the COPERNICA (P20_01224) and METAMORFOSIS (FEDER_US-1381375) projects, both supported by the Junta de Andalucía.

2015). Some examples thereof are inconsistencies between the knowledge base and a set of test cases (Felfernig, Friedrich, Jannach, et al. 2004; Le, Felfernig, Uta, Benavides, et al. 2021), or inconsistencies between user requirements and the knowledge base (Felfernig, Friedrich, Schubert, et al. 2009). In such scenarios, *diagnosis detection mechanisms* are essential to identify *preferred minimal sets* of constraints (i.e., diagnoses) that are less important (from the user’s point of view) and can be adapted or deleted to restore consistency in the knowledge base (Reiter 1987; Felfernig, Schubert, and Zehentner 2012).

Direct diagnosis techniques have been recognized as efficient solutions in identifying faulty constraints without predetermining the corresponding conflict sets (Felfernig, Schubert, and Zehentner 2012; Felfernig, Walter, et al. 2018). FASTDIAG (Felfernig, Schubert, and Zehentner 2012) is a typical example of these techniques, designed to find *one preferred minimal diagnosis* at a time within a given set of constraints (C). The algorithm divides C into two subsets. If a subset is consistent, then diagnosis detection must not be applied to this subset since no diagnosis elements can be found in it. This way, C can be reduced by half, and the algorithm returns one *preferred minimal diagnosis* at a time. Although FASTDIAG works efficiently in many scenarios, there exist cases where it faces runtime issues, especially in *interactive settings*, where users are interacting with a configurator with a huge and complex knowledge base and expect to receive instant responses (Felfernig, Walter, et al. 2018).

Consistency checking is an expensive computational task that makes up most of FASTDIAG’s execution time (Felfernig, Hotz, et al. 2014). A practical solution for this issue is to pre-calculate in parallel consistency checks potentially required by FASTDIAG. This solution provides fast answers for consistency checks (via simple lookup in a list of already-calculated consistency checks instead of a direct solver call), which helps to accelerate the algorithm’s execution. Based on this idea, we propose a novel diagnosis detection algorithm, so-called FASTDIAGP, dealing with FASTDIAG’s run-time limitation. FASTDIAGP is a parallelized version of FASTDIAG, adopting the *speculative programming principle* (Burton 1985) to pre-calculate consistency checks. Although this principle is not new, modern CPUs with integrated parallel computation capabilities now make it possible to implement some speculative approaches.

Our contributions are *three-fold*. *First*, we show how to parallelize direct diagnosis based on a flexible *look-ahead strategy* to scale its performance depending on the number of available computing cores. *Second*, we show how to integrate the proposed approach into FASTDIAG, which is applicable to interactive constraint-based applications. *Third*, using the inconsistent *Linux-2.6.33.3* feature model taken from Diverso Lab’s benchmark (Heradio et al. 2022), we show the performance improvements of FASTDIAGP when working with large-scale configuration knowledge bases. Particularly, this algorithm improves the performance of diagnosis detection tasks scaling with available CPU cores, making it possible to efficiently solve more complex diagnosis problems.

4.2 Related Work

Solution Search. The increasing size and complexity of knowledge bases have led to the need of improving search solutions (Bordeaux, Hamadi, and Samulowitz 2009; Gent et al. 2018). Such solutions have been implemented as par-

allelization algorithms in different contexts. For example, (Bordeaux, Hamadi, and Samulowitz 2009) propose a parallelization approach to determine solutions for sub-problems independent of available cores. Due to the development of multi-core CPU architectures, parallelization approaches have become increasingly popular to exploit computing resources better and obtain expected results more efficiently.

Conflict Detection. Determining more efficiently minimal conflicts is a core requirement in many application settings (Jannach, Schmitz, and Shchekotykhin 2015). In constraint-based reasoning scenarios, the QUICKXPLAIN algorithm (Junker 2004) is applied to identify minimal conflict sets following the divide-and-conquer-based strategy. Although this algorithm helps to reduce the number of needed consistency checks significantly and supports interactive settings, it faces runtime performance issues. In this context, (Vidal et al. 2021) proposes a conflict detection approach based on speculative programming, called the so-called PARALLELIZED QUICKXPLAIN. The empirical results show that this approach helps to significantly improve the runtime performance of QUICKXPLAIN.

Conflict Resolution. Conflict detection is the basis of *conflict resolution* that attempts to identify sets of minimal diagnoses (Reiter 1987; Marques-Silva, Heras, et al. 2013). For instance, (Jannach, Schmitz, and Shchekotykhin 2015; Jannach, Schmitz, and Shchekotykhin 2016) propose approaches to parallelize the computation of hitting sets (diagnoses). In these studies, a level-wise expansion of a breadth-first search tree is adopted to parallelize model-based diagnosis (Reiter 1987) and compute minimal cardinality diagnoses. However, the determination of individual conflict sets is still a sequential process (based on QUICKXPLAIN (Junker 2004)). In another study, (Jannach, Schmitz, and Shchekotykhin 2016) replace the level-wise expansion with a full hitting set parallelization and take into account additional mechanisms to ensure diagnosis minimality. Although the mentioned approaches focus on the parallelization of conflict resolution, they do not offer solutions to increase the efficiency of conflict detection. In this chapter, based on the speculative programming principle (Burton 1985), we propose an algorithm integrating a parallelized conflict resolution mechanism that helps to significantly improve the runtime performance of direct diagnosis processes.

4.3 Example Configuration Knowledge Base

For demonstration purposes, we introduce a working example with a configuration knowledge base from the *smartwatch* domain. A *Smartwatch* must have at least one type of *Connector* and *Screen*. The connector can be one or more out of the following: *GPS*, *Cellular*, *Wifi*, or *Bluetooth*. The screen can be either *Analog*, *High Resolution*, or *E-ink*. A *Smartwatch* may include a *Camera* and a *Compass*. Besides, *Compass* requires *GPS* and *Camera* requires *High Resolution*. Finally, *Cellular* and *Analog* exclude each other.

Our simplified configuration knowledge base can be represented as a *configuration task* which is defined as a *constraint satisfaction problem* (CSP) (Rossi,

Beek, and Walsh 2006). A *configuration task* and its *configuration* (solution) are defined as follows (Hotz et al. 2014):

Definition 4.1 (Configuration task). A configuration task can be defined as a CSP (V, D, C) where $V = \{v_1, v_2 \dots v_n\}$ is a set of variables, $D = \{dom(v_1), dom(v_2) \dots dom(v_n)\}$ is a set of domains for each of the variables in V , and $C = C_{KB} \cup C_R$ is a set of constraints restricting possible solutions for a configuration task. C_{KB} represents the configuration knowledge base (the configuration model) and C_R represents a set of user requirements.

Definition 4.2 (Configuration). A configuration (solution) S for a given configuration task (V, D, C) is an *assignment* $A = \{v_1 = a_1 \dots v_n = a_n\}$, $a_i \in dom(v_i)$. S is **valid** if it is *complete* (i.e., each variable in V has a value) and *consistent* (i.e., S fulfills the constraints in C).

Example 4.1 (CSP-based representation of a *Smartwatch* configuration task). A CSP-based representation of a configuration task $(V, D, C = C_{KB} \cup C_R)$ that can be generated from our simplified configuration knowledge base is the following (see Table 4.1 for constraints in C_{KB} and C_R):

- $V = \{\text{Smartwatch}, \text{Connector}, \text{Screen}, \text{Camera}, \text{Compass}, \text{GPS}, \text{Cellular}, \text{Wifi}, \text{Bluetooth}, \text{Analog}, \text{High Resolution}, \text{E-ink}\},$
- $D = \{dom(\text{Smartwatch}) \dots dom(\text{E-ink})\}, \text{ where } dom(v_i) = \{(t)rue, (f)alse\},$
- $C_{KB} = \{c_0 \dots c_9\}, C_R = \{c_{10} \dots c_{13}\}.$

According to Table 4.1, we can observe that some constraints in C_R are inconsistent with the constraints in C_{KB} . For instance, c_{10} and c_{11} in C_R are inconsistent with c_9 in C_{KB} . Therefore, no solution can be found for this configuration task. For related faulty constraints, see Example 4.2.

Due to inconsistent constraints in the knowledge base/user requirements, the reasoning engine (e.g., constraint solver) cannot determine a solution. In this context, identifying explanations (in terms of diagnoses) is extremely important to help users adapt their requirements and thus restore consistency. In the next section, we introduce basic concepts regarding diagnoses and preferred minimal diagnoses. Also, we revisit the FASTDIAG algorithm (Felfernig, Schubert, and Zehentner 2012) and show how a preferred minimal diagnosis can be determined using this algorithm.

4.4 Determination of Preferred Diagnoses

Since the notions of a (minimal) *conflict* and a (minimal) *diagnosis* will be used in the following sections, we provide the corresponding definitions here. We use $consistent(C)$ to denote that the constraint set C is consistent, and $inconsistent(C)$ to denote that the constraint set C is inconsistent.

A *conflict set* can be defined as a minimal set of constraints that is responsible for an inconsistency, i.e., a situation in which no solution can be found for a given set of constraints C (see Definition 4.3).

Table 4.1: Constraints in $C_{KB} = \{c_0 \dots c_9\}$ derived from our simplified configuration knowledge base, $C_R = \{c_{10} \dots c_{13}\}$ is a set of user requirements.

CSP representation	
Constraints in the knowledge base - C_{KB}	
c_0	$Smartwatch = t$
c_1	$Smartwatch \leftrightarrow Connector$
c_2	$Smartwatch \leftrightarrow Screen$
c_3	$Camera \rightarrow Smartwatch$
c_4	$Compass \rightarrow Smartwatch$
c_5	$Connector \leftrightarrow (GPS \vee Cellular \vee Wifi \vee Bluetooth)$
c_6	$Screen \leftrightarrow xor(Analog, High Resolution, E-ink)$
c_7	$Camera \rightarrow High Resolution$
c_8	$Compass \rightarrow GPS$
c_9	$\neg(Cellular \wedge Analog)$
User requirements - C_R	
c_{10}	$Cellular = t$
c_{11}	$Analog = t$
c_{12}	$Compass = t$
c_{13}	$GPS = f$

Definition 4.3 (Conflict set). A *conflict set* is a set $CS \subseteq C : inconsistent(CS)$. CS is *minimal* iff $\nexists CS' \subset CS : inconsistent(CS')$.

Example 4.2 (Minimal conflict sets). We are able to identify the following minimal conflict sets: $CS_1 = \{c_{10}, c_{11}\}$ and $CS_2 = \{c_{12}, c_{13}\}$. The minimality property is fulfilled since $\nexists CS_3 : CS_3 \subset CS_1$ and $\nexists CS_4 : CS_4 \subset CS_2$.

In order to resolve all conflicts, we need to determine corresponding hitting sets (also denoted as diagnoses (Reiter 1987)) that have to be adapted or deleted to make the user requirements consistent with the knowledge base. Based on the definition of a conflict set, we now introduce the definition of a *diagnosis task* and a corresponding *diagnosis*.

Definition 4.4 (Diagnosis task). A diagnosis task can be defined by a tuple (C_R, C_{KB}) , where C_R is a set of user requirements to be analyzed and C_{KB} is a set of constraints specifying the configuration knowledge base.

Definition 4.5 (Diagnosis and Maximal Satisfiable Subset). A *diagnosis* Δ of a diagnosis task (C_R, C_{KB}) is a set $\Delta \subseteq C_R : consistent(C_R \setminus \Delta \cup C_{KB})$. Δ is *minimal* iff $\nexists \Delta' \subset \Delta : consistent(C_R \setminus \Delta' \cup C_{KB})$. A complement of Δ (i.e., $C_R \setminus \Delta$) is denoted as *Maximal Satisfiable Subset* (MSS) Ω .

Example 4.3 (Minimal diagnoses). Applying conflict-directed diagnosis approaches (Reiter 1987) to the diagnosis task $(C_R = \{c_{10} \dots c_{13}\}, C_{KB} = \{c_0 \dots c_9\})$ presented in Examples 4.1 and 4.2, the corresponding minimal diagnoses are the following: $\Delta_1 = \{c_{10}, c_{12}\}$, $\Delta_2 = \{c_{10}, c_{13}\}$, $\Delta_3 = \{c_{11}, c_{12}\}$, and $\Delta_4 = \{c_{11}, c_{13}\}$.

4.4.1 Preferred Diagnosis

To resolve given inconsistencies, a user has to choose a diagnosis consisting the constraints that need to be adapted/deleted. In this context, a diagnosis less important to the user is chosen first (Junker 2004). Such a diagnosis is a so-called “*preferred diagnosis*” (Marques-Silva and Previti 2014) (defined in Definition 4.8 based on Definitions 4.6 and 4.7).

Definition 4.6 (Strict total order). Let $<$ be a *strict total order* over the constraints in $C = \{c_1 \dots c_m\}$ which is represented as $\langle c_1 < c_2 < \dots < c_m \rangle$, i.e., c_i is preferred over c_{i+1} .

Definition 4.7 (Anti-lexicographic preference, A-Preference). Given a *strict total order* $<$ over C , a set $X \subseteq C$ is *anti-lexicographically* preferred over another set $Y \subseteq C$ (denoted $X <_{antilex} Y$) iff $\exists_{i \leq k \leq m} : c_k \in Y \setminus X$ and $X \cap \{c_{k+1} \dots c_m\} = Y \cap \{c_{k+1} \dots c_m\}$.

Definition 4.8 (Preferred diagnosis). A minimal diagnosis Δ for a given diagnosis task (C_R, C_{KB}) is a preferred diagnosis for (C_R, C_{KB}) iff $\forall \Delta' : \Delta' <_{antilex} \Delta$.

Given a strict total order $<$ over a set of constraints, there exists a unique preferred diagnosis.

Example 4.4 (A preferred diagnosis). Given two minimal diagnoses $\Delta_3 = \{c_{11}, c_{12}\}$, $\Delta_4 = \{c_{11}, c_{13}\}$ and the strict total order $\langle c_{11} < c_{12} < c_{13} \rangle$, we can say:

- Δ_3 is *anti-lexicographically* preferred over Δ_4 since $c_{13} \in \Delta_4 \setminus \Delta_3$ with $\Delta_4 \cap \emptyset = \Delta_3 \cap \emptyset$.
- Δ_4 is a preferred diagnosis since it contains c_{13} that is less important than c_{12} presented in Δ_3 .

4.4.2 FASTDIAG

FASTDIAG (Felfernig, Schubert, and Zehentner 2012) determines a diagnosis without the need of conflict detection and a related derivation of hitting sets (Reiter 1987). Algorithms 4.1 and 4.2 below show a variant of FASTDIAG, where Algorithm 4.2 - FD determines a *maximal satisfiable subset* Ω instead of a *minimal correction subset* as in the original version presented in (Felfernig, Schubert, and Zehentner 2012).

Algorithm 4.1 – FASTDIAG includes two variables C and B , where C consists of potentially faulty constraints in C_R and B contains correct constraints in C_{KB} . The constraint ordering in C conforms to the definition of the *strict total order* (see Definition 4.6). If *inconsistent*($B \cup C$), then Algorithm 4.2 – FD is activated to identify constraints in C that are responsible for the inconsistency. FD determines an MSS Ω , from which the corresponding minimal diagnosis can be derived ($\Delta = C \setminus \Omega$). In FD, if *consistent*($B \cup C$), then C is returned since no diagnosis elements can be found and C becomes part of the MSS. If there is only one constraint c_i in C , then c_i is an element of a conflict since *inconsistent*($B \cup C$). This element is removed from C (by returning an empty set) to guarantee that C is an MSS. If *inconsistent*($B \cup C$) and C has more

Algorithm 4.1 FASTDIAG(C, B) : Δ

```

1: if  $C = \emptyset$  or CONSISTENT( $B \cup C$ ) then
2:   return  $\emptyset$ 
3: else
4:   return  $C \setminus \text{FD}(C, B, \emptyset)$ 
5: end if

```

Algorithm 4.2 FD($C = \{c_1 \dots c_n\}, B, \rho$) : Ω

```

1: if  $\rho \neq \emptyset$  and CONSISTENT( $B \cup C$ ) then
2:   return  $C$ 
3: end if
4: if  $|C| = 1$  then
5:   return  $\emptyset$ 
6: end if
7: SPLIT( $C, C_l, C_r$ )
8:  $\Omega_2 \leftarrow \text{FD}(C_l, B, C_r)$ 
9:  $\Omega_1 \leftarrow \text{FD}(C_r, B \cup \Omega_2, C_l \setminus \Omega_2)$ 
10: return  $\Omega_1 \cup \Omega_2$ 

```

than one element, the SPLIT function is called to divide C into two subsets $C_l = \{c_1 \dots c_k\}$ and $C_r = \{c_{k+1} \dots c_n\}$, where $k = \lfloor \frac{n}{2} \rfloor$. Finally, FASTDIAG returns an MSS and the corresponding minimal diagnosis Δ .

The parameter ρ in the FD algorithm plays an important role in avoiding redundant consistency checks. Assigning C_r to ρ (see *line 8*) triggers a consistency check for $B \cup C_l$. If *consistent*($B \cup C_l$), C_l will be returned by the FD call at *line 2*. The FD call at *line 9* does not trigger a consistency check in *line 1* since $C_l \setminus \Omega_2 = \emptyset$, i.e., $B \cup C_r \cup \Omega_2 = B \cup C_r \cup C_l = B \cup C$ has been already checked. The details of how FASTDIAG works are shown in Figure 4.1 on the basis of our working example.

4.5 FASTDIAGP – Parallelized FASTDIAG

General idea. FASTDIAGP is the parallelized version of FASTDIAG, in which we integrate a *look-ahead* mechanism adopting the speculative programming principle (Burton 1985) into the CONSISTENT function. The *look-ahead* mechanism performs two tasks: (1) anticipating potential consistency checks that FASTDIAG might need in the near future, and (2) scheduling the asynchronous execution of anticipated consistency checks.

To ensure correct and useful anticipated consistency checks (i.e., FD will request consistency checks' results in its next calls), the anticipation of the *look-ahead* mechanism complies with the two following principles *P1* and *P2*:

- *P1* (Following two assumptions concerning the consistency of $B \cup C$): In each recursive step of FD, the decision for the next recursive call depends on the consistency of the current $B \cup C$. If *inconsistent*($B \cup C$), FD applies the *divide-and-conquer* strategy to C . Otherwise, a strategy for ρ is used, holding the sibling half of C . Thus, an *inconsistency* assumption helps to discover the next level of the FD execution tree, while a *consistency* one helps to exploit the

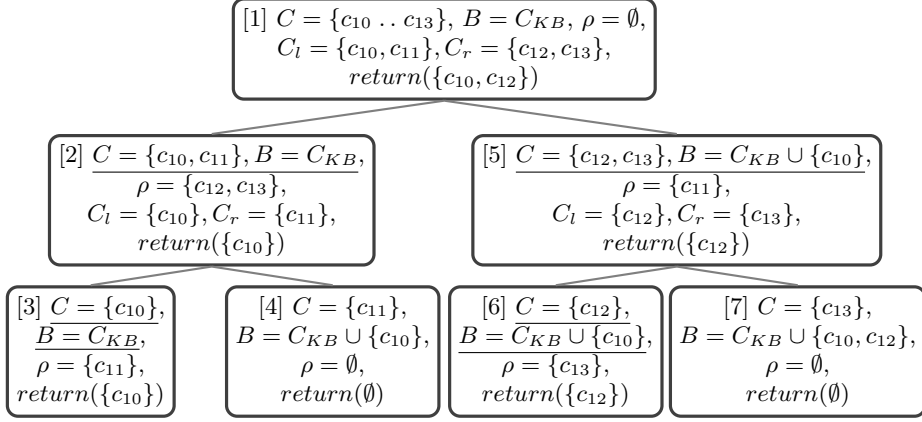


Figure 4.1: FD execution trace for $C = \{c_{10} \dots c_{13}\}$ and $B = C_{KB}$. FD determines a MSS $\Omega = \{c_{10}, c_{12}\}$ and the corresponding diagnosis $\Delta = \{c_{11}, c_{13}\}$. The underlined C and B instances denote FD consistency checks.

sibling of the current call. This way, the *look-ahead* mechanism can generate all needed consistency checks without redundancy.

- *P2 (Complying with the divide-and-conquer strategy of FASTDIAG)*: In each recursive step of FD, when $\text{inconsistent}(B \cup C)$ and C is not a singleton, consistency checks for the two halves of C are triggered by FD. Thus, regarding the *look-ahead* mechanism, when the current consideration set is not a singleton, the *divide-and-conquer* strategy is applied to both *consistency* and *inconsistency* assumption branches to obtain the same effect as FD can make. The current consideration set could be C or ρ .

Besides, the anticipation considers the computer resources in terms of available CPU cores ($\#cores$) in order to generate adequate consistency checks. For instance, the current FD execution needs the consistency check for $C = \{c_1, c_2\}$. The algorithm also knows that $B = \emptyset$, $\rho = \{c_3, c_4\}$, i.e., the remaining constraints will be checked if C is consistent, and the system has a 4-cores CPU. In this context, the *look-ahead* mechanism can generate and execute in parallel three consistency checks:

1. $C_1 = \{c_1, c_2\}$ - the consistency check, which is being required by FD.
2. $C_2 = \{c_1\}$ - the first half of C , which will be checked in the next FD call if $\{c_1, c_2\}$ is inconsistent.
3. $C_3 = \{c_1, c_2, c_3\}$ - a union of C and the first half of ρ , which will be checked if $\{c_1, c_2\}$ is consistent.

Since the *look-ahead* mechanism runs on one CPU core, only three future consistency checks are generated in our example. Each generated consistency check is asynchronously executed in one core.

LOOKUP table. Consistency checks generated by the *look-ahead* mechanism are stored in a global LOOKUP table (see Table 4.2). If FD needs to know the consistency of a given set of constraints, a simple lookup is triggered to get the corresponding consistency check's result. Assume that there is no con-

Table 4.2: A LOOKUP table created in our working example, including part of consistency checks generated by ADDCC in the LOOKAHEAD function (see Figure 4.2) and executed in parallel. The ‘ $\neg$ ’ entries indicate that the corresponding consistency checks are still ongoing or have not been started.

node-id	constraint set	consistent
1	$\{C_{KB} \cup \{c_{10} \dots c_{13}\}\}$	<i>false</i>
1.2	$\{C_{KB} \cup \{c_{10}, c_{11}\}\}$	<i>false</i>
1.2.1	$\{C_{KB} \cup \{c_{10}, c_{11}, c_{12}\}\}$	–
1.2.1.2	$\{C_{KB} \cup \{c_{10}, c_{11}, c_{13}\}\}$	–
1.2.2	$\{C_{KB} \cup \{c_{10}\}\}$	<i>true</i>

Algorithm 4.3 CONSISTENT(C, B, ρ) : *Boolean*

```

1: if  $\neg$ EXISTCC( $B \cup C$ ) then
2:   LOOKAHEAD( $C, B, \{\rho\}$ )
3: end if
4: return LOOKUP( $B \cup C$ )

```

sistency check for the requested set in the LOOKUP table. In that case, the algorithm runs the *look-ahead* mechanism to generate and execute in parallel anticipated consistency checks. Consistency checks in the LOOKUP table can also be exploited to restrict the generation of the consistency checks that have already been created in the previous steps of the *look-ahead* mechanism. This way, all anticipated consistency checks can be done only once and will not waste computer resources.

CONSISTENT function. FASTDIAGP uses the CONSISTENT function (see Algorithm 4.3) that requires three inputs: a consideration set C , a background knowledge B , and a set of constraints ρ that has not been checked yet. Different from the CONSISTENT function in FASTDIAG, the additional parameter ρ is needed to help the *look-ahead* mechanism conduct inferences about future needed consistency checks. Since these sets are FD’s inputs at each recursive step, no additional computations are required.

The CONSISTENT function checks the existence of a consistency check for $B \cup C$ in the LOOKUP table. If this is the case, the function returns the consistency check’s outcome. Otherwise, it activates the LOOKAHEAD function (see Algorithm 4.4) to generate further consistency checks that might be relevant in upcoming FD recursive calls.

LOOKAHEAD function. The *look-ahead* mechanism is implemented in the recursive LOOKAHEAD function (Algorithm 4.4), requiring three parameters: (1) a consideration set C , (2) a background set B holding the already-considered and *assumed* consistent constraints, and (3) an ordered set ϕ in which each item is a set of constraints to be considered when C is a singleton or assumed to be consistent.

The first constraint set ϕ_1 of ϕ is always the most recent second subset divided from C in the last recursive call. ϕ_1 has to be considered first when

Algorithm 4.4 LOOKAHEAD($C, B, \phi = \{\{\phi_1\} \dots \{\phi_p\}\}$)

```

1: if  $curGCC < maxGCC$  then
2:   if  $\neg \text{EXISTCC}(B \cup C)$  then
3:      $curGCC \leftarrow curGCC + 1$ 
4:      $\text{ADDCC}(B \cup C)$ 
5:   end if
6:    $\{B \cup C \text{ assumed consistent}\}$ 
7:   if  $|\phi| > 1 \wedge |\phi_1| = 1 \wedge \text{EXISTCC}(B \cup C \cup \phi_1)$  then {case 1.1}
8:      $\text{SPLIT}(\phi_2, \phi_{2l}, \phi_{2r})$ 
9:      $\text{LOOKAHEAD}(\phi_{2l}, B \cup C, \phi_{2r} \cup (\phi \setminus \{\phi_1, \phi_2\}))$ 
10:  else if  $\phi \neq \emptyset \wedge |\phi_1| = 1$  then {case 1.2}
11:     $\text{LOOKAHEAD}(\phi_1, B \cup C, \phi \setminus \phi_1)$ 
12:  else if  $\phi \neq \emptyset \wedge |\phi_1| > 1$  then {case 1.3}
13:     $\text{SPLIT}(\phi_1, \phi_{1l}, \phi_{1r})$ 
14:     $\text{LOOKAHEAD}(\phi_{1l}, B \cup C, \phi_{1r} \cup (\phi \setminus \phi_1))$ 
15:  end if
16:   $\{B \cup C \text{ assumed inconsistent}\}$ 
17:  if  $|C| > 1$  then {case 2.1}
18:     $\text{SPLIT}(C, C_l, C_r)$ 
19:     $\text{LOOKAHEAD}(C_l, B, C_r \cup \phi)$ 
20:  else if  $|C| = 1 \wedge |\phi_1| = 1$  then {case 2.2}
21:     $\text{LOOKAHEAD}(\phi_1, B, \phi \setminus \phi_1)$ 
22:  else if  $|C| = 1 \wedge |\phi_1| > 1$  then {case 2.3}
23:     $\text{SPLIT}(\phi_1, \phi_{1l}, \phi_{1r})$ 
24:     $\text{LOOKAHEAD}(\phi_{1l}, B, \phi_{1r} \cup (\phi \setminus \phi_1))$ 
25:  end if
26: end if

```

LOOKAHEAD takes into account sets of ϕ . With the structure of ϕ , the order of consistency checks generated by LOOKAHEAD matches the order of consistency checks requested by FASTDIAG. For instance, in node [1.2.2] of the LOOKAHEAD execution trace (see Figure 4.2), ϕ contains two sets: $\phi_1 = \{c_{11}\}$ and $\phi_2 = \{c_{12}, c_{13}\}$. The set $\phi_2 = \{c_{12}, c_{13}\}$, which is added to ϕ in node [1], is the second half of $C = \{c_{10} \dots c_{13}\}$. The set $\phi_1 = \{c_{11}\}$ separated from $C = \{c_{10}, c_{11}\}$ is added to ϕ in node [1.2]. Since ϕ_1 is added later on, it will be considered first in the next LOOKAHEAD call. In particular, it is considered in node [1.2.2.2] before taking into account ϕ_2 in node [1.2.2.2.1]. This mechanism works in a similar fashion as FASTDIAG.

In each LOOKAHEAD call, two global parameters $curGCC$ (initialized by zero) and $maxGCC$ (initialized by $\#cores$) are used to restrict the maximum number of generated speculative consistency checks. LOOKAHEAD checks the available space for further consistency checks and examines if any consistency check exists for $B \cup C$ so far. If not, then the ADDCC function is called to activate a consistency check for $B \cup C$ asynchronously and adds an entry of $B \cup C$ to the LOOKUP table. Next, LOOKAHEAD predicts potentially relevant consistency checks needed by FD based on *two assumptions*: (1) $B \cup C$ is consistent and (2) $B \cup C$ is inconsistent (see the details in the next paragraphs). The order of two assumptions is opposite to this in FASTDIAG since consistency

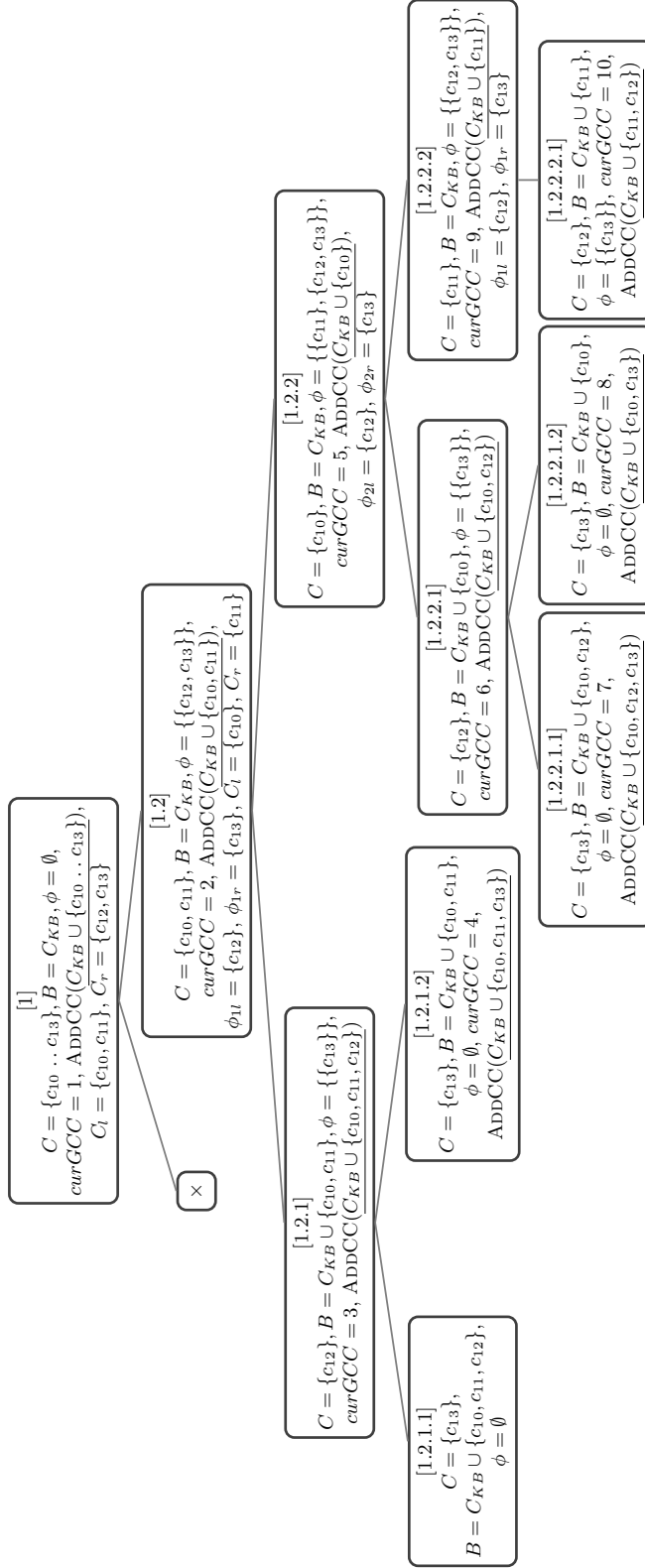


Figure 4.2: LOOKAHEAD execution trace for $C = \{c_{10} \dots c_{13}\}$, $B = C_{KB}$, $\phi = \emptyset$, and $maxGCC = 10$. The consistency checks $\{C_{KB} \cup \{c_{10}, c_{11}\}\}$, $\{C_{KB} \cup \{c_{10}\}\}$, $\{C_{KB} \cup \{c_{10}, c_{12}\}\}$, and $\{C_{KB} \cup \{c_{10}, c_{12}, c_{13}\}\}$ generated by LOOKAHEAD, can be used by the FASTDIAG execution trace illustrated in Figure 4.1.

checks with larger cardinality should be executed in advance, which helps reduce the waiting time in case the corresponding consistency checks are still ongoing.

Assumption 1 ($B \cup C$ is consistent): The function further checks the sets of ϕ when $B \cup C$ is consistent, i.e., all LOOKAHEAD calls will have $B \cup C$ as the background set.

- If there is a consistency check for $B \cup C \cup \phi_1$ in the LOOKUP table (see *case 1.1* in Algorithm 4.4), LOOKAHEAD omits ϕ_1 and further checks ϕ_2 . The function considers the first half of ϕ_2 (ϕ_{2l}) as the consideration set. $\{\phi_1, \phi_2\}$ in ϕ are replaced with the second half of ϕ_2 (i.e., $\phi_{2r} \cup (\phi \setminus \{\phi_1, \phi_2\})$). Let's have a look at an example in Figure 4.2. In node [1.2.2], the consistency check for $C_{KB} \cup \{c_{10}, c_{11}\}$ has already been generated. Hence, in node [1.2.2.1], the function omits $\phi_1 = \{c_{11}\}$ and proceeds with a further look ahead for ϕ_2 .
- If $|\phi_1| = 1$ (see *case 1.2* in Algorithm 4.4), the input parameters of the function are ϕ_1 , $B \cup C$, and $\phi \setminus \phi_1$. In our example (Figure 4.2), this case will be applied to expand the *consistent* branch of node [1.2.2.1].
- If $|\phi_1| > 1$ (see *case 1.3* in Algorithm 4.4), ϕ_1 is divided into two halves ϕ_{1l} and ϕ_{1r} . LOOKAHEAD is called where C is replaced with the first half ϕ_{1l} and ϕ_1 in ϕ is replaced with the second half ϕ_{1r} (i.e., $\phi_{1r} \cup (\phi \setminus \phi_1)$). One example is shown in node [1.2] of Figure 4.2.

Assumption 2 ($B \cup C$ is inconsistent): Consistency checks for the halves of C are necessary to identify elements of C responsible for the inconsistency.

- If the cardinality of C is greater than 1 (see *case 2.1* in Algorithm 4.4), C is divided into two halves C_l and C_r . Thereafter, LOOKAHEAD for C_l is called, where $B \cup C_l$ becomes the next consistency check, and C_r is stored in ϕ to be considered when $B \cup C_l$ is assumed to be consistent or C_l is a singleton. Note that C_r is added to the head of ϕ to be considered first when the function takes into account the sets of ϕ . Examples of this case are shown in nodes [1] and [1.2] of Figure 4.2.
- If $|C| = 1$, the function further checks the sets of ϕ with the *inconsistency* assumption:
 - If the first set of ϕ is a singleton ($|\phi_1| = 1$) (see *case 2.2* in Algorithm 4.4), LOOKAHEAD is called, where C is replaced with ϕ_1 and ϕ_1 is removed from ϕ . The execution of LOOKAHEAD according to this case can be found in nodes [1.2.1] and [1.2.2] of Figure 4.2.
 - If ϕ_1 consists of several constraints to be considered ($|\phi_1| > 1$) (see *case 2.3* in Algorithm 4.4), ϕ_1 is divided into two halves ϕ_{1l} and ϕ_{1r} . The function LOOKAHEAD is called, where the first half ϕ_{1l} is the consideration set, ϕ_1 in ϕ is replaced with ϕ_{1r} (i.e., $\phi_{1r} \cup (\phi \setminus \phi_1)$). In Figure 4.2, the *inconsistent* branch of node [1.2.2.2] is created based on this case.

4.6 Theoretical Analysis of FASTDIAGP

Soundness and Completeness of FASTDIAGP. FASTDIAGP preserves the soundness and completeness properties of FASTDIAG. FASTDIAGP does not change the FD function but integrates the *look-ahead* mechanism, where some consistency checks requested by FD are pre-calculated. Besides, FD performs simple lookups instead of expensive solver calls. Therefore, FASTDIAGP obtains better performance, and the returned diagnosis is minimal and preferred.

Soundness and Completeness of LOOKAHEAD. LOOKAHEAD generates correct consistency checks for subsets of the consideration set C . Assuming that C' is a set generated by LOOKAHEAD, but $C' \not\subseteq (C \cup \phi)$. We can say that $\exists c_k : c_k \in C' \wedge c_k \notin (C \cup \phi)$. Since LOOKAHEAD does not exploit constraints outside of $C \cup \phi$, C' is not a set generated by LOOKAHEAD. Besides, by following two principles *P1* and *P2*, LOOKAHEAD can generate all possible combinations of constraints in C .

Uniqueness of anticipated consistency checks. LOOKAHEAD exploits ϕ in both assumption branches, which leads to redundant consistency checks when exploiting ϕ according to the *consistency* assumption. The checks in *lines 2* and *7* (Algorithm 4.4) assure that generated consistency checks are unique.

Complexity analysis. *FD complexity.* The *worst-case* complexity of FD in terms of the number of needed consistency checks for determining MSS Ω and the corresponding diagnosis $\Delta = C - \Omega$ is $2d \times \log_2(\frac{n}{d}) + 2d$, where d is the *set size of the minimal diagnosis*, n is the *number of constraints* in C , and $2d$ represents the branching factor and the number of leaf-node consistency checks. The *best-case* complexity is $\log_2(\frac{n}{d}) + 2d$. In the worst case, each diagnosis element is located in a different path of the search tree. The factor $\log_2(\frac{n}{d})$ represents the depth of a path of the FD search tree. In the best case, all constraints part of a diagnosis are included in a single path of the search tree.

LOOKAHEAD complexity. Assuming that $\phi = \emptyset$, the number of consistency checks (N) generated by LOOKAHEAD is the sum of all possible combinations of n constraints in the consideration set C . It means that $N = \sum_{i=1}^n \binom{n}{i} = 2^n - 1$. Due to the uniqueness of LOOKAHEAD, the upper bound of its space complexity in terms of the number of LOOKAHEAD calls is $2^n - 1$.

Termination of LOOKAHEAD. If $\max GCC \leq 2^n - 1$, recursive calls of LOOKAHEAD stop when $\max GCC$ consistency checks are generated. Otherwise, LOOKAHEAD terminates if C and ϕ are empty.

4.7 Empirical Evaluation

Experiment design. In this study, we compared the performance of FASTDIAGP and FASTDIAG according to three aspects: (1) *run-time* R needed to determine the *preferred diagnosis*, (2) *speedup* S that tells us the gain we get through the parallelization, and (3) *efficiency* E representing the ratio between the speedup and the number of processes in which we run the algorithm. In particular, *speedup* S_p is computed as $= T_1/T_p$, where T_1 is the wall time when

4.7. Empirical Evaluation

Table 4.3: *Average runtime R (in sec), speedup S , and efficiency E of FAST-DIAGP ($\#cores > 1$) versus FASTDIAG ($\#cores = 1$) needed for determining the preferred diagnosis with a repetition rate of 3 per setting and $maxGCC = \#cores - 1$. $|diag|$ denotes the cardinality of the diagnosis.*

		#cores				
		1	4	8	16	32
1	R	4.56	3.08	2.77	2.63	3.29
	S		1.48	1.65	1.74	1.39
	E		0.49	0.24	0.12	0.05
2	R	5.60	4.00	3.69	3.71	5.05
	S		1.40	1.52	1.51	1.11
	E		0.47	0.22	0.10	0.04
4	R	8.13	5.95	5.76	6.43	10.11
	S		1.37	1.41	1.26	0.80
	E		0.46	0.20	0.08	0.03
8	R	11.96	9.06	8.74	9.63	14.52
	S		1.32	1.37	1.24	0.82
	E		0.44	0.20	0.08	0.03
16	R	20.95	16.80	16.38	19.02	29.28
	S		1.25	1.28	1.10	0.72
	E		0.42	0.18	0.07	0.02

using 1 core (FASTDIAG) and T_p is the wall time when p cores are used. The efficiency E_p is defined as S_p/p . These aspects were analyzed in two dimensions: the *diagnosis cardinality* and the *available computing cores* ($\#cores$).

Dataset and Procedure. The basis for these evaluations was the *Linux-2.6.33.3* configuration knowledge base taken from Diverso Lab’s benchmark² (Heradio et al. 2022). The characteristics of this knowledge base are the following: $\#features = 6,467$; $\#relationships = 6,322$; and $\#cross-tree constraints = 7,650$. For this knowledge base, we randomly synthesized³ and collected 20,976 inconsistent sets of requirements, whose cardinality ranges from 5 to 250. We applied systematic sampling technique (Mostafa and Ahmad 2018) to select 10 inconsistent requirements with diagnosis cardinalities of 1, 2, 4, 8, and 16.

The diagnosis algorithms were implemented in *Python* using SAT4J (Le Berre and Parrain 2010) as a reasoning solver.⁴ We used the CNF class of PySAT

²<https://github.com/diverso-lab/benchmarking>

³To ensure the reproducibility of the results, we used the seed value of 141982L for the random number generator.

⁴The dataset, the implementation of algorithms, and evaluation programs can be found at <https://github.com/AIG-ist-tugraz/FastDiagP>.

Table 4.4: *Average runtime (in sec) of FASTDIAGP ($\#cores > 1$) versus FASTDIAG ($\#cores = 1$) needed for determining the preferred diagnosis with a repetition rate of 3 per setting and $maxGCC = 7$. $|diag|$ denotes the cardinality of the preferred diagnosis. The *bold values* prove the optimal number of CPU cores ($\#cores = 8$).*

		<i>#cores</i>			
		1	8	16	32
$ diag $	1	4.56	2.78	2.79	2.81
	2	5.60	3.69	3.70	3.72
	4	8.13	5.76	5.80	5.87
	8	11.96	8.74	8.78	8.84
	16	20.95	16.38	16.41	16.64

(Ignatiev, Morgado, and Marques-Silva 2018) for representing constraints and the *Python multiprocessing* package for running parallel tasks. All experiments reported in the chapter were conducted with an Amazon EC2 instance⁵ of the type c5a.8xlarge, offering 32 vCPUs with 64-GB RAM.

Results. The experimental results show that FASTDIAGP outperforms the sequential direct diagnosis approach in almost all scenarios (see the **bold values** in Table 4.3). Besides, in Tables 4.3 and 4.4, the optimal number of CPU cores is 8 and the corresponding speedup values range from 1.28 to 1.65, showing the runtime deduction up to 40%. The $\#cores$ higher than 8 becomes less efficient for boosting the performance. Particularly, in Table 4.3, the increase of $\#cores$ (that also triggers the increase of $maxGCC$ ($maxGCC = \#cores - 1$)) leads to gradual runtime increase (i.e., lower performance). A parallelization mechanism with more than 8 cores is not so much helpful in such a scenario. This manifests when $\#cores = 32$, $maxGCC = 31$, and $|diag| = 4, 8, 16$. The reason is that the LOOKAHEAD function applies a sequential mechanism. When $maxGCC$ gets higher, the runtime of LOOKAHEAD increases exponentially, leading to a significant increase of FASTDIAGP’s runtime. Besides, Table 4.4 confirms that the utilization of more than 8 CPU cores becomes less efficient. In this evaluation, our idea was to fix the $maxGCC$ value ($maxGCC = 7$) to see how the performance of FASTDIAGP is when $\#cores$ is higher than 8.

4.8 Conclusions

In this chapter, we have proposed a parallelized variant of the FASTDIAG algorithm to diagnose over-constrained problems. Our parallelized approach helps to exploit multi-core architectures and provides an efficient preferred diagnosis detection mechanism. Furthermore, our approach is helpful for dealing with complex over-constrained problems, boosting the performance of various knowledge-based applications, and making these systems more accessible, especially in the

⁵<https://aws.amazon.com/ec2/instance-types/c5/>

context of interactive settings. Open topics for future research are the following: (1) performing more in-depth evaluations on the basis of industrial configuration knowledge bases (in this context, we plan to analyze the different *look-ahead* search approaches, e.g. breadth-first search, in further detail), and (2) applying speculative reasoning for supporting anytime diagnosis tasks.

Chapter 5

DIRECTDEBUG: Automated Testing and Debugging of Feature Models

This chapter is based on the results documented in (Le, Felfernig, Uta, Benavides, et al. 2021). The author of this thesis carried out the literature research, algorithm design, implementation, empirical evaluation, and data analysis, and contributed to the writing of the paper.

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 chapter, we introduce DIRECTDEBUG which is a direct diagnosis approach to the *automated testing and debugging of variability models*.¹ 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.

5.1 Introduction

Feature models support the representation of variability and commonality properties of software artifacts (Kang et al. 1990; Benavides, Segura, and Ruiz-Cortés 2010). 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* (Kang et al. 1990) support the representation of hierarchies including cross-tree constraints such as *excludes* and *requires* relationships. *Cardinality-based feature models* (Czarnecki, Helsen, and Eisenecker 2005) extend basic ones

¹The work presented in this chapter has been partially funded by the EU Horizon 2020 OPENREQ project (732463), the Austrian Research Promotion Agency PARXCEL project (880657), and the EU FEDER program MINECO OPHELIA project (RTI2018-101204-B-C22).

with cardinalities (> 1) of feature relationships. Finally, *extended feature models* (Batory 2005) 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 (Zeller 2001; Trinidad et al. 2008; Benavides, Felfernig, et al. 2013). 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 chapter is organized as follows. After a discussion of related work (Section 5.2), we introduce an example of a feature model (Section 5.3). 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 5.4) and debugging (Section 5.5) of feature models. In Section 5.6, we report initial results of a performance analysis of our approach. The chapter is concluded with Section 5.7.

5.2 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. (Benavides, Segura, and Ruiz-Cortés 2010).

Conflict Detection. A conflict set (conflict) can be regarded as a *subset that induces an inconsistency*. For example, Zeller et al. (Zeller and Hildebrandt 2002) 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 (Junker 2004) 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 (Reiter 1987; Felfernig, Schubert, and Zehentner 2012)) 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 (Zeller and Hildebrandt 2002; Junker 2004), the approach presented in this chapter 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. (Bakker et al. 1993). In this line of research, Trinidad et al. (Trinidad et al. 2008) show

how to determine such diagnoses in the context of inconsistent feature models. In contrast to the work of Bakker et al. (Bakker et al. 1993) and Trinidad et al. (Trinidad et al. 2008), the approach presented in this chapter 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. (Felfernig, Friedrich, Jannach, et al. 2004) who introduce a model-based diagnosis approach (Reiter 1987) to resolve conflicts in knowledge bases induced by test cases. Compared to Felfernig et al. (Felfernig, Friedrich, Jannach, et al. 2004), our approach is based on *direct diagnosis* (no conflict detection needed) which allows for an efficient determination of diagnoses (Felfernig, Schubert, and Zehentner 2012).

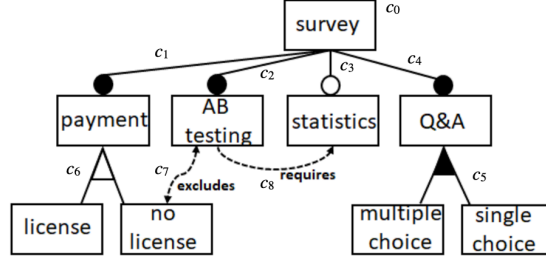
Diagnosis for Reconfiguration. A reconfiguration can be regarded as a *set of adaptations of feature settings in a changed configuration that are needed to restore consistency between the configuration and the corresponding feature model* (Felfernig, Walter, et al. 2018). Using a constraint-based representation (Tsang 1993) of a feature model, White et al. (White et al. 2010) show how to apply the concepts of model-based diagnosis (Reiter 1987) 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 chapter generalizes the concepts of Trinidad et al. (Trinidad et al. 2008) and White et al. (White et al. 2010) 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* (Felfernig, Schubert, and Zehentner 2012) is to significantly improve the performance of hitting set based diagnosis approaches (Reiter 1987) by supporting the calculation of diagnoses without the need of predetermining conflict sets. In this chapter, we show how to support the automated testing and debugging of feature models using direct diagnosis (Felfernig, Schubert, and Zehentner 2012).

The *contributions* of this chapter are threefold. *First*, we show how to extend direct diagnosis (Felfernig, Schubert, and Zehentner 2012) 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.

5.3 Feature Model Semantics

Feature models are used to represent software variability properties by specifying features and their relationships in a hierarchical fashion (Kang et al. 1990). Features are arranged in a hierarchical fashion with one specific root feature f_r which has to be included in every configuration (Benavides, Segura, and Ruiz-Cortés 2010). 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 (Batory 2005).

Figure 5.1: An example of a (presumably faulty) *survey software* feature model.

Feature Model Semantics. For the discussions in this chapter, we follow the feature model representation of Benavides et al. (Benavides, Segura, and Ruiz-Cortés 2010) 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) (Tsang 1993). Each feature f_i is related to the binary domain $\{(t)rue, (f)alse\}$. The mentioned relationships and cross-tree constraints are represented as constraints on the CSP level.

Example Feature Model. Figure 5.1 depicts an example of a *presumably faulty feature model* (for details, see Section 5.4) 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. (Benavides, Segura, and Ruiz-Cortés 2010).

- **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 5.1, 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 (Benavides, Segura, and Ruiz-Cortés 2010).
- **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 5.1, *statistics* is an optional feature connected to *survey*.
- **Alternative:** exactly one feature f_i out of $\{f_1 \dots f_k\}$ has to be selected if feature f_a has been selected. On the logical level, *alternative* relationships can be formalized as follows: $f_1 = t \leftrightarrow (f_2 = f \wedge \dots \wedge f_k = f \wedge f_a =$

$t) \wedge \dots \wedge f_k = t \leftrightarrow (f_1 = f \wedge \dots \wedge f_{k-1} = f \wedge f_a = t)$. An example thereof is *payment* with the subfeatures *license* and *nolicense*.

- **Or:** at least one feature f_i out of a feature set $\{f_1 \dots f_k\}$ has to be selected if feature f_a has been selected. Relationships of type *or* can be formalized as follows: $f_a \leftrightarrow f_1 = t \vee f_2 = t \vee \dots \vee f_k = t$. An example of a feature f_a is *Q&A*, the subfeatures are *multiplechoice* and *singlechoice*.
- **Requires:** feature f_b must be included in a configuration if feature f_a 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 $\neg(f_a \wedge f_b)$. An example of an *excludes* relationship is: $\neg(ABtesting \wedge nolicense)$.

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 5.1).

Definition 5.1 (Configuration Task). A configuration task (F, D, C) is defined by a feature set $F = \{f_1, f_2 \dots f_n\}$ and a set of feature domains $D = \{dom(f_1), dom(f_2) \dots dom(f_n)\}$ ($dom(f_i) = \{(t)true, (f)alse\}$). Furthermore, $C = CR \cup CF$ represents constraints restricting the possible solutions for a configuration task where $CR = \{c_1, c_2 \dots c_k\}$ is a set of user requirements and $CF = \{c_{k+1}, c_{k+2} \dots c_m\}$ a set of feature model constraints.

In Definition 5.1, CR is an additional set of constraints which specifies which features should be included in a configuration.

Based on Definition 5.1, we introduce the concept of a *configuration* (solution) for a configuration task (Definition 5.2).

Definition 5.2 (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 5.1 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.

- $F = \{survey, payment, license, nolicense, ABtesting, statistics, Q\&A, multiplechoice, singlechoice\}$
- $D = \{dom(survey) = \{t, f\} \dots dom(singlechoice) = \{t, f\}\}$
- $CF = \{c_0 : survey = t, c_1 : survey \leftrightarrow payment, c_2 : survey \leftrightarrow ABtesting, c_3 : statistics \rightarrow survey, c_4 : survey \leftrightarrow Q\&A, c_5 : Q\&A \leftrightarrow multiplechoice \vee singlechoice, c_6 : (license \leftrightarrow \neg nolicense \wedge payment) \wedge (nolicense \leftrightarrow \neg license \wedge payment)\}, c_7 : \neg(ABtesting \wedge nolicense), c_8 : ABtesting \rightarrow statistics\}$

5.4. Testing Feature Models

Table 5.1: Example positive test cases $T_\pi = \{t_1 \dots t_4\}$.

ID	Test Case (Constraint)
t_1	$nolicense = t$
t_2	$license = t \wedge statistics = f$
t_3	$payment = f$
t_4	$singlechoice = f$

- $CR = \{c_9 : license = t, c_{10} : ABtesting = t\}$

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

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

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* (Felfernig, Schubert, and Zehentner 2012).

5.4 Testing Feature Models

In Figure 5.1, *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_π specifies the *intended behavior* of a knowledge base (feature model). Positive test cases $t_i \in T_\pi$ 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 5.1).

Avoiding *nolicense* to be a *dead feature* can be achieved by a positive test case $t_1 : nolicense = t$ (*nolicense* should be included at least one configuration). Similarly, an example of a partial *survey* configuration could be: $t_2 : license = t \wedge statistics = f$. 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 $ABtesting \wedge license \rightarrow statistics$ as a test case. In the following, we will integrate $\{t_1 \dots t_4\}$ in our discussion of a diagnosis algorithm that supports the automated testing and debugging of feature models.

Negative Test Cases. Negative test cases $t_i \in T_\Theta$ 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 5.5).

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 (Benavides, Segura, and Ruiz-Cortés 2010) (e.g., to avoid *dead features* f_i , a test case $f_i = t$ can be generated for each feature).

5.5 Automated Debugging with DIRECTDEBUG

A diagnosis (Δ) includes exactly those constraints responsible for the faulty behavior of a feature model. Intuitively, constraints in Δ have to be deleted or adapted to make the feature model consistent with T_π (see Definition 5.3). Thus, a diagnosis helps a feature model engineer to focus diagnosis search. To enable such a functionality, test cases are needed.

Definition 5.3 (Diagnosis and Maximal Satisfiable Subset). Given a feature model with a set CF of feature model constraints and a set of positive test cases $T_\pi = \{t_1, t_2 \dots t_l\}$. A diagnosis $\Delta = \{c_1, c_2 \dots c_q\}$ 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. Δ is minimal iff $\neg \exists \Delta' \subset \Delta$ such that $\forall t_i \in T_\pi : \{t_i\} \cup CF - \Delta'$ consistent. A complement of Δ (i.e., $CF - \Delta$) is denoted as *Maximal Satisfiable Subset* (MSS Γ).

Example Diagnoses. The minimal diagnoses that can be derived in our working example are depicted in Table 5.2. There are two options for restoring the consistency between the feature model (Figure 5.1) and $\{t_1 \dots t_4\}$: either delete/adapt the constraints in Δ_1 or do this with the constraints of Δ_2 . For example, Δ_1 suggests to take a look at c_1 (reconsider the relationship between *payment* and *survey*) and c_2 (reconsider the relationship between *ABtesting* and *survey*).

Table 5.2: Example diagnoses $\Delta_1 = \{c_1, c_2\}$ and $\Delta_2 = \{c_1, c_7, c_8\}$.

Diagnosis	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
Δ_1	×	×	—	—	—	—	—	—
Δ_2	×	—	—	—	—	—	×	×

Algorithm 5.1 DIRECTDEBUG($\delta, C = \{c_1 \dots c_n\}, B, T_\pi$) : Γ

```

1:  $T'_\pi \leftarrow T_\pi$ 
2: if  $\delta \neq \emptyset \wedge \text{IsConsistent}(C \cup B, T_\pi, T'_\pi)$  then
3:   return  $C$ 
4: end if
5: if  $|C| = 1$  then
6:   return  $\emptyset$ 
7: end if
8:  $k = \lfloor \frac{n}{2} \rfloor$ 
9:  $C_1 \leftarrow c_1 \dots c_k; C_2 \leftarrow c_{k+1} \dots c_n;$ 
10:  $\Gamma_2 \leftarrow \text{DIRECTDEBUG}(C_1, C_1, B, T'_\pi);$ 
11:  $\Gamma_1 \leftarrow \text{DIRECTDEBUG}(C_1 - \Gamma_2, C_2, B \cup \Gamma_2, T'_\pi);$ 
12: return  $\Gamma_1 \cup \Gamma_2$ 

```

Diagnosis Approach. DIRECTDEBUG determines minimal diagnoses directly (Felfernig, Schubert, and Zehentner 2012), i.e., without predetermining conflicts. It extends direct diagnosis with test cases (T_π). DIRECTDEBUG is activated with the diagnosis candidates C (constraints of CF considered as potentially faulty) and the background knowledge B (constraints of CF assumed to be consistent and correct). The algorithm (Algorithm 1) determines an MSS Γ , the corresponding minimal diagnosis is $C - \Gamma$.

The constraint c_0 should not be diagnosable, since empty feature models should not be allowed (this would be the case if c_0 is part of a diagnosis). We assume $c_0 : \text{survey} = t$ to be part of the *background knowledge* B which consists of constraints assumed to be correct, i.e., B entails those constraints which should not be regarded as diagnosis candidates. Furthermore, B includes those *negative test cases* $t_i \in T_\Theta$ in *negated form* which are consistent with $C \cup B$. In our example, we assume $T_\Theta = \emptyset$ for simplicity. Before starting DIRECTDEBUG, all t_i in $T_\pi = \{t_1 \dots t_q\}$ and $T_\Theta = \{t_{q+1} \dots t_z\}$ 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_π are forwarded to DIRECTDEBUG which are inconsistent with $C \cup B$. In our setting, the original set of positive test cases $T_\pi = \{t_1 \dots t_4\}$ is reduced to $\{t_1 \dots t_3\}$ since t_4 does not induce an inconsistency in $C \cup B$, i.e., there exists a configuration (solution) for $C \cup B \cup \{t_4\}$.

DIRECTDEBUG determines a maximal satisfiable subset MSS(Γ) (Definition 5.3) where $C \subseteq CF - \{c_0\}$ (consideration set), $B = CF - C \cup \{c_0\} \cup T^-$ (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_π that are inconsistent with $C \cup B$. $C \subseteq CF - \{c_0\}$

Table 5.3: Runtimes (in *msec*) of DIRECTDEBUG with different constraint set (CF) and test set (T_π) cardinalities ($|T_\Theta| = 0$) evaluated on an Intel Core i7 (6 cores) 2.60GHz with 16 GB of RAM.

$ T_\pi $	$ CF $					
	10	20	50	100	500	1000
5	0.1	0.4	1.1	2.4	24.8	114.0
10	0.2	0.5	1.8	4.3	35.4	170.4
25	0.6	1.3	4.3	11.7	108.1	375.5
50	1.0	2.3	7.9	22.7	241.6	695.5
100	1.9	4.5	14.7	38.8	449.0	1251.5
250	6.6	11.2	32.4	90.4	1191.3	2877.9
500	20.3	24.2	60.3	158.2	2338.1	5680.1

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_π . Obviously, test cases have to be checked individually, i.e., each activation of ISCONSISTENT results in $|T_\pi|$ constraint solver activations (consistency checks). ISCONSISTENT returns t (*true*) if every test case in T_π is consistent with $C \cup B$, otherwise f (*false*). Only test cases inducing an inconsistency with $C \cup B$ are stored (returned) in T'_π (the remaining inconsistent positive test cases). A consistency check is only activated if $\delta \neq \emptyset$ ($\delta = \emptyset$ indicates that C has already been checked for consistency with B).

DIRECTDEBUG Execution. An execution trace of DIRECTDEBUG is shown in Figure 5.2. 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 inconsistent($\{t_j\} \cup C \cup B$), then c_i is considered a part of a diagnosis Δ . If $C \cup B$ is consistent with the test cases in T_π , C is returned since no diagnosis elements can be found in C .

Diagnosis Determination. DIRECTDEBUG returns a *maximal satisfiable subset* ($\Gamma = \{c_2 \dots c_6\}$) (see Figure 5.2). To determine a minimal diagnosis Δ , we have to determine the MSS-complement, i.e., $C - \Gamma$ which is $\{c_1, c_7, c_8\}$.

5.6 Performance Analysis

To evaluate the performance of DIRECTDEBUG,² we synthesized test feature models (see Table 5.3). For model synthesis, we applied the BETTY generator (Segura et al. 2012) using the parameters *#test positive cases* ($|T_\pi|$, with a

²The test dataset used for evaluation purposes can be found here: <https://github.com/AIG-ist-tugraz/DirectDebug>.

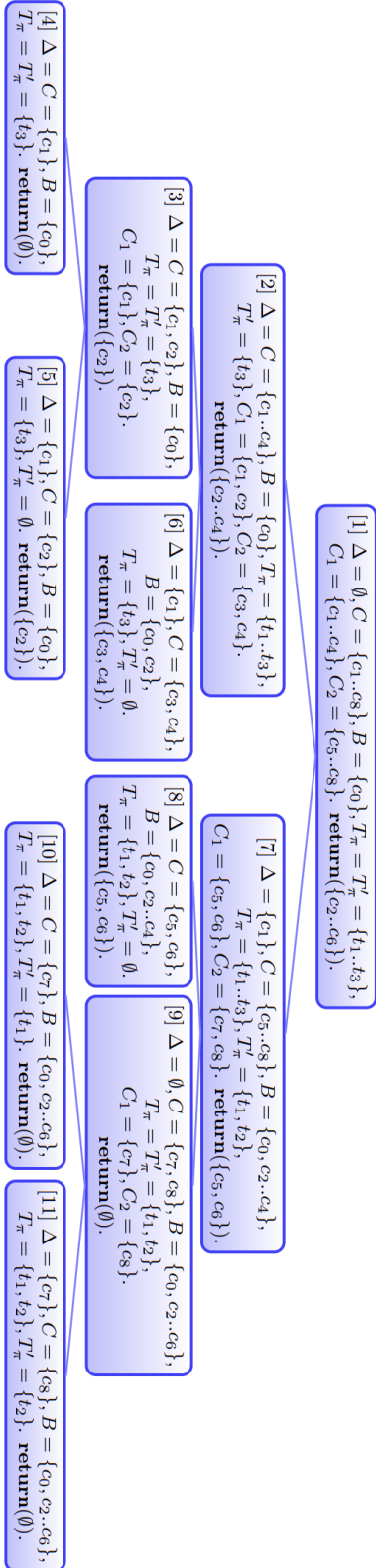


Figure 5.2: DIRECTDEBUG execution trace for $C = \{c_1 \dots c_8\}$, $B = \{c_0\}$, and $T_\pi = \{t_1 \dots t_3\}$. Since $C \cup B \cup \{t_4\}$ is consistent, there is no need to further analyze t_4 . DIRECTDEBUG determines a maximal satisfiable subset $\text{MSS}(T) = \{c_2 \dots c_6\}$, the corresponding diagnosis is $\Delta = \{c_1, c_7, c_8\}$ (the MSS complement).

30% share of inconsistency-inducing test cases) and *#constraints in CF* ($|CF|$, where $\#variables = \frac{|CF|}{2}$) with $C = CF - \{c_0\}$. Since each test case check needs a constraint solver call,³ runtimes increase with an increasing number of test cases and constraints. Each $|T_\pi| \times |CF|$ entry in Table 5.3 represents the average DIRECTDEBUG (diagnosis) computing time after 3 repetitions.

5.7 Conclusions

We have introduced an automated testing and debugging approach for feature models. Test cases are used to induce conflicts in the feature model knowledge base. Using *direct diagnosis*, we show how consistency can be restored using a minimal diagnosis. Thus, we can pro-actively support feature model designers and can expect significant time savings in feature model development and evolution. Major issues for future work include the generation of test cases taking into account, for example, different coverage metrics. Furthermore, we will use (feature) model quality metrics to better predict the most relevant diagnoses. Finally, we will continue our evaluations with real-world feature models.

³For evaluation purposes, we used choco-solver.org.

Chapter 6

WIPEOUTR: Automated Redundancy Detection for Feature Models

This chapter is based on the results documented in (Le, Felfernig, Uta, Tran, et al. 2022). The author of this thesis carried out the literature research, algorithm design, implementation, empirical evaluation, and data analysis, and contributed to the writing of the paper.

Feature models are used to specify variability and commonality properties of software artifacts. In order to assure high-quality models, different feature model analysis and testing operations can be applied. In this chapter, we present two new algorithms that help to make feature model configuration as well as different kinds of analysis operations more efficient. Specifically, we focus on the *automated identification of redundancies in feature models and corresponding test suites*. *Redundant constraints* in feature models can lead to low-performing configuration (solution) search and also to additional efforts in feature model debugging. *Redundant feature model test cases* can trigger inefficiencies in testing operations. In this chapter, we introduce WIPEOUTR which is an algorithmic approach to support the automated identification of redundancies.¹ This approach has the potential to significantly improve the quality of feature model development and configuration.

6.1 Introduction

Feature models represent variability aspects of software systems (Kang et al. 1990). *Basic feature models* (Kang et al. 1990) allow the representation of hierarchical feature relationships as well as cross-tree constraints specifying model properties orthogonal to hierarchical relationships. *Cardinality-based feature models* further extend the expressiveness by allowing the inclusion of cardinality specifications. Finally, *extended feature models* allow the inclusion of feature attributes (Batory 2005).

¹The work presented in this chapter has been developed within the scope of the PARXCEL project, which is funded by the Austrian Research Promotion Agency (880657).

With the increasing size and complexity of feature models, the development and maintenance of these models can be an error-prone and time-consuming task leading to situations where the search and repair of erroneous feature model constraints becomes a challenge (Benavides, Felfernig, et al. 2013; Kowal, Ananieva, and Thüm 2016). It can be the case that a feature model does not completely represent the intended variability properties of the application domain (Zeller 2001; Trinidad et al. 2008; Benavides, Felfernig, et al. 2013). The reasons can be missing domain knowledge leading to faulty modeling, cognitive overheads as a result of feature model size and complexity, and pre-existing and outdated parts of a feature model.

As a consequence, feature model development and maintenance has to be supported by intelligent techniques that help to identify model parts that should be analyzed in more detail. One possibility to achieve this goal is to apply model-based analysis approaches such as DELTA DEBUGGING (Zeller and Hildebrandt 2002) and DIRECTDEBUG (Le, Felfernig, Uta, Benavides, et al. 2021) which help to identify minimal sets of model elements that are responsible for an unintended model behavior. In this chapter, we focus on a somewhat orthogonal aspect which is the *automated identification of redundancies* in feature models. Redundant model elements (constraints) can decrease model understandability, lead to increased development and maintenance efforts, and also decrease the efficiency of constraint solving (Kowal, Ananieva, and Thüm 2016). In the context of feature models, a redundancy is interpreted as a set of model elements that can be deleted from the model without changing the semantics of the original model, i.e., the solution space in terms of the set of possible feature model configurations remains the same.

The major contributions are the following. *First*, we introduce an algorithm that helps to identify redundant model elements in feature models, i.e., to determine a maximum set of model elements (represented as constraints) that can be deleted from the feature model without changing the semantics of the underlying model. *Second*, we introduce an algorithm that helps to determine feature model test cases that can be omitted due to redundancy, i.e., there is no need to take into account such test cases. *Third*, we present the results of a performance analysis with a real-world feature model. Overall, related evaluation results show that the runtime performance of both, feature model testing as well as feature model configuration, can be improved.

The remainder of this chapter is organized as follows. In Section 6.2, we provide an overview of related work. Thereafter, we introduce a feature model from the domain of *survey software configuration* (see Section 6.3). In this context, we also sketch how to represent feature models as constraint satisfaction problems (CSPs). Exactly this representation is used in Section 6.4 to introduce algorithms supporting the identification of redundancies in feature models and test suites. In Section 6.5, we report initial results of a performance analysis of our approach. The chapter is concluded with a discussion of future research in Section 6.6.

6.2 Related Work

The existing state-of-the-art in feature model quality assurance can be summarized as follows.

Feature Model Analysis. Feature model (FM) analysis operations help to assure different expected properties of feature models. Examples thereof are the *non-existence of dead features* (each feature should be active in at least one possible configuration), the *non-existence of void feature models* (there should exist at least one configuration for each feature model), and the *non-existence of false-optional features* (for each feature f there should exist at least one configuration where f is deactivated). A detailed discussion of FM analysis operations is provided in Benavides et al. (Benavides, Segura, and Ruiz-Cortés 2010) and Kowal et al. (Kowal, Ananieva, and Thüm 2016). Also, in the software engineering context, the identification of irrelevant features can contribute to reduce the overall effort in software testing by reducing combinatorial complexity (Kim et al. 2011). Furthermore, in the context of analyzing so-called featured transition systems, for example, *false optional transitions* may indicate a redundancy that can be removed (Beek et al. 2022).

Feature Model Testing and Debugging. Concepts of feature model testing and debugging help to identify those elements of a feature model that can be regarded as responsible for an unintended (faulty) semantics of a feature model. In this context, feature model testing means to test if a set of test cases (intended properties of a feature model) are fulfilled. If this is not the case, corresponding (automated) debugging functions help to identify minimal sets of responsible feature model elements. In automated debugging, i.e., the identification of constraints that are contradictory with real-world feature relationships, feature model analysis operations are often regarded as *test cases* (Le, Felfernig, Uta, Benavides, et al. 2021). For example, the identification of *void features* is a typical example of a feature model analysis operation (Benavides, Segura, and Ruiz-Cortés 2010; Kowal, Ananieva, and Thüm 2016). Similarly, this property can be interpreted as a type of test case $f_i = \text{true}$, i.e., a (positive) test case representing an intended semantics that there should exist at least one configuration that supports $f_i = \text{true}$ (Le, Felfernig, Uta, Benavides, et al. 2021). If we do not want to allow void feature models, the corresponding test case could be defined as $f_r = \text{true}$ where f_r represents the root feature of the feature model. Finally, the test case $f_i = \text{false}$ can be used to specify that a specific feature f_i should not be false optional.

Handling of Conflicts. Conflicts can be regarded as sets of model elements that are responsible for a faulty (unintended) behavior. For example, the QUICKXPLAIN algorithm (Junker 2004) can be applied to identify minimal subsets of constraints of a constraint satisfaction problem (CSP) (Rossi, Beek, and Walsh 2006) that are responsible for an inconsistency, i.e., no solution can be identified for the CSP. In a similar line of research, Zeller et al. (Zeller and Hildebrandt 2002) present the DELTA DEBUGGING algorithm which basically supports the identification of property subsets in test cases responsible for the faulty behavior of a software component. Having identified a set of conflicts, diagnosis algorithms can be applied for conflict resolution – see, for example, (Bakker et al. 1993; Reiter 1987; Trinidad et al. 2008; Felfernig, Walter, et al. 2018; Le, Felfernig, Uta, Benavides, et al. 2021; Le, Felfernig, Tran, Atas, et al. 2021). In this context, diagnoses can be seen as *deletion subsets* that indicate options for restoring model consistency.

Identification of Redundancies. The identification of redundancies is important in different application scenarios (Piette 2008). Informally, a constraint can be regarded as redundant if its deletion does not change the semantics of the knowledge base, i.e., the corresponding solution space remains the same. More formally, if $C = \{c_1 \dots c_m\}$ is a set of constraints and $c_\alpha \in C$ is redundant, this means that $C - \{c_\alpha\} \cup \{\neg c_\alpha\}$ remains inconsistent (the solution space of $C - \{c_\alpha\}$ is equivalent to C). In other words, if $c_\alpha \in C$ is redundant, c_α *logically follows* from $C - \{c_\alpha\}$, i.e., $C - \{c_\alpha\} \models c_\alpha$.

Piette (Piette 2008) introduces an incomplete algorithm for the identification of redundant clauses in the context of SAT solving, i.e., the identification of redundancies cannot be guaranteed. Sabin and Freuder (Sabin and Freuder 1999) propose a similar approach in the context of solving conditional CSPs where redundancies are not analyzed in the domain knowledge but in the context of configuration search space exploration. Redundancies in FMs are analyzed, for example, in Felfernig et al. (Felfernig, Benavides, et al. 2013) where a formal definition of redundancy is given based on the idea of interpreting redundancy identification as a specific type of conflict resolution in knowledge bases. Kowal et al. (Kowal, Ananieva, and Thüm 2016) provide an in-depth analysis of different types of feature model analysis operations. In this context, redundancy in feature models is characterized in terms of the concept of semantic (solution space) equivalence, i.e., the deletion of redundant model elements does not change the feature model semantics.

A definition of redundancy in the context of linear optimization is introduced in (Paulraj and Sumathi 2010; Fukada, Gärtner, and Szedlak 2015). The presented algorithms focus on specific properties of linear programs and are not directly applicable to the logical structures of CSPs generated out of, for example, feature models. The assumption of the existence of a larger set of redundant constraints is known as the *high redundancy assumption* – this can be the case if a group of engineers is in charge of model development. In contrast, the *low redundancy assumption* is related to scenarios where single engineers are responsible for these tasks (Grimm et al. 2011).

Compared to existing research on redundancy detection in software models (specifically variability models), we introduce two new algorithms that support (1) the identification of all redundant constraints in a feature model and (2) the identification of redundant test cases in a feature model test suite. In this context, we introduce complete algorithms for redundancy detection.

6.3 Semantics of Feature Models

Features (represented as nodes) in a feature model are typically organized hierarchically where it is assumed that the root feature f_r is included (activated) in every configuration. Relationships between features are used to represent hierarchical constraints as well as so-called cross-tree constraints. In the example feature model of this chapter (see Figure 6.1), we use the basic modeling concepts discussed a.o. in Benavides et al. (Benavides, Segura, and Ruiz-Cortés 2010).

The mentioned relationships include the hierarchical constraints *mandatory*, *optional*, *alternative*, or and two types of cross-tree constraints which are *requires* and *excludes*. For analysis purposes, feature models can be trans-

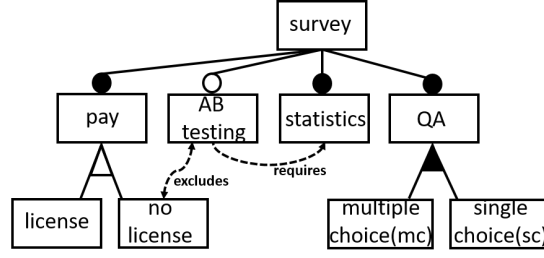


Figure 6.1: An example of a *survey software* feature model (Le, Felfernig, Uta, Benavides, et al. 2021) with a simple example of a redundant constraint (*ABtesting* requires *statistics*). Since *statistics* must be part of every feature model configuration, the mentioned requires-relationship can be regarded as redundant.

lated into corresponding CSPs (Rossi, Beek, and Walsh 2006) where each feature can be either activated or deactivated, i.e., the domain of each feature is $\{(t)true, (f)alse\}$. Figure 6.1 shows an example *feature model* from the domain of *software services for survey management*. For example, the feature *statistics* represents a statistics component enabling the analysis of user answers in surveys.

- **Mandatory:** a feature f_b is regarded as *mandatory* if its inclusion is mandatory depending on the inclusion of feature f_a ($f_a \leftrightarrow f_b$). For example, feature *Q&A* is mandatory (see Figure 6.1) which means that it has to be part of every configuration.
- **Optional:** if f_b is optional, it may be included if f_a is included in a configuration ($f_b \rightarrow f_a$). For example, *ABtesting* is an optional feature connected to *survey* (see Figure 6.1).
- **Alternative:** exactly one f_i out of $\{f_1 \dots f_k\}$ must be selected if f_a is selected ($f_1 = t \leftrightarrow (f_2 = f \wedge \dots \wedge f_k = f \wedge f_a = t) \wedge \dots \wedge f_k = t \leftrightarrow (f_1 = f \wedge \dots \wedge f_{k-1} = f \wedge f_a = t)$). An example of an alternative feature is *pay* including the subfeatures *license* and *nolicense*.
- **Or:** at least one out of $\{f_1 \dots f_k\}$ must be selected if the parent feature has been selected ($f_a \leftrightarrow f_1 = t \vee f_2 = t \vee \dots \vee f_k = t$). An example such of a parent feature f_a is *Q&A* where *mc* and *sc* are the corresponding subfeatures.
- **Requires:** f_b has to be included if f_a is included in a configuration ($f_a \rightarrow f_b$), for example, $ABtesting \rightarrow statistics$.
- **Excludes:** Two features f_a and f_b must not be combined (f_a excludes feature f_b and vice versa). On a logical level, this can be defined as $\neg(f_a \wedge f_b)$. An example of an *excludes* relationship is $\neg(ABtesting \wedge nolicense)$.

A feature model represents variability properties of configurable artifacts (e.g., software components). In order to be able to support model quality assurance tasks such as those discussed in Section 6.2, we need to introduce a

formal definition of a feature model configuration task and a corresponding solution (configuration). In the following, we assume that a configuration task is represented as a CSP and a solution in terms of a set of variable instantiations indicating feature inclusion or exclusion.

Definition 6.1 (Feature Model Configuration Task). A feature model (FM) configuration task (F, D, C) can be defined by a set $F = \{f_1, f_2 \dots f_n\}$ of features and corresponding domains $D = \{dom(f_1), dom(f_2) \dots dom(f_n)\}$ ($dom(f_i) = \{(t)true, (f)alse\}$). The set of constraints $C = CF \cup CR$ is composed of a set of domain constraints restricting the set of possible solutions (CF) and a set of user (customer) requirements (CR) specifying specific preferences of the current user that have to be taken into account by the final configuration. In this context, $CF = \{c_1 \dots c_k\}$ and $CR = \{c_{k+1} \dots c_m\}$.

After having specified a feature model configuration task, a constraint solver can be activated to find a corresponding solution (feature model configuration).

Definition 6.2 (Feature Model Configuration). Given an FM configuration task, a feature model configuration is an assignment A of the given feature variables $f_i \in F$. A is regarded as *consistent* if A does not violate any constraint in $CR \cup CF$ and *complete* if every feature has an assignment describing inclusion or exclusion.

Example CSP. A constraint satisfaction problem (CSP) representing a feature model configuration task $(F, D, C = CF \cup CR)$ derived from the feature model in Figure 6.1 is the following. Please note that $c_0 : survey = t$ is a *root constraint* which prevents the generation of empty configurations, i.e., configurations where no feature is activated.

- $F = \{survey, pay, license, nolicense, ABtesting, statistics, Q\&A, multiplechoice(mc), singlechoice(sc)\}$
- $D = \{dom(survey) = \{t, f\} \dots dom(sc) = \{t, f\}\}$
- $CF = \{c_0 : survey = t, c_1 : survey \leftrightarrow pay, c_2 : ABtesting \rightarrow survey, c_3 : survey \leftrightarrow statistics, c_4 : survey \leftrightarrow Q\&A, c_5 : Q\&A \leftrightarrow mc \vee sc, c_6 : (license \leftrightarrow \neg nolicense \wedge pay) \wedge (nolicense \leftrightarrow \neg license \wedge pay), c_7 : \neg(ABtesting \wedge nolicense), c_8 : ABtesting \rightarrow statistics\}$
- $CR = \{c_9 : license = t, c_{10} : ABtesting = t\}$

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

- $\{survey = t, pay = t, license = t, nolicense = f, ABtesting = t, statistics = t, Q\&A = t, mc = t, sc = t\}$

Testing Feature Models. In order to support feature model quality assurance, test suites (T) can be applied to define the intended behavior of a feature model (Le, Felfernig, Uta, Benavides, et al. 2021). Examples of test cases t_i (see also Table 6.1) in such test suites are $t_1 : ABtesting = t$ (assuring that there exists at least one configuration with *ABtesting* activated)

Table 6.1: Example set of test cases specifying the intended behavior of the feature model (see Figure 6.1).

ID	constraint
t_1	$ABtesting = t$
t_2	$statistics = f$
t_3	$ABtesting = t \wedge statistics = t$

and $t_2 : statistics = f$ (assuring that there exists at least one configuration with *statistics* deactivated). A further test case could specify that there should exist at least one co-occurrence of the features *ABtesting* and *statistics*, i.e., $t_3 : ABtesting = t \wedge statistics = t$. When taking a more detailed look at the test cases t_1 and t_3 , one might detect that t_3 is more specific than t_1 which means that $t_3 \models t_1$, i.e., t_1 logically follows from t_3 and can thus be regarded as redundant (t_1 is already checked by t_3). In this case, t_1 could be removed from the test set and thus help to make corresponding testing and debugging processes more efficient.

Definition 6.3 (Redundancy-free Test Suite). A feature model test suite $T = \{t_1 \dots t_q\}$ is a set of test cases t_i (represented as arbitrary CSP constraints) that have to be individually fulfilled by the feature model (corresponding CSP). A test suite is *redundancy-free* iff $\neg \exists \{t_i, t_j\} \subseteq T (i \neq j) : t_i \models t_j$.

Test Cases for Feature Models. Test cases can be derived from already *completed feature model configurations*. Furthermore, test cases are often *specified by domain experts*. Finally, test cases can be specified on the basis of analysis operations (Benavides, Felfernig, et al. 2013; Kowal, Ananieva, and Thüm 2016). For example, to avoid *dead features* f_i in a feature model, a test case $f_i = t$ can be generated for each individual feature of a feature model.

Following the idea of redundancy-free test suites, we now introduce the definition of a redundancy-free feature model.

Definition 6.4 (Redundancy-free Feature Model). Given an FM configuration task with the constraints $CF = \{c_1 \dots c_k\}$, the model is redundancy-free iff $\neg \exists c_i \in CF : CF - c_i \models c_i$.

Note that this definition refers to the domain constraints CF of the feature model, i.e., does not include the user requirements, however – if needed – the same idea can also be applied to the given user requirements (CR). In our CSP, constraint c_8 can be regarded as redundant, since *statistics* is defined as a mandatory feature (constraint c_3).

In the following, we show how to identify maximal sets of constraints of a feature model that can (could) be deleted from the feature model without changing the model semantics, i.e., the solution space remains the same. Furthermore, we show how to identify redundant test cases that can be deleted from a test suite used for feature model quality assurance.

6.4. Redundancy Detection

Algorithm 6.1 WIPEOUTR_T($T = \{t_1 \dots t_k\}$)

```

1:  $T_\pi \leftarrow T; T_\Delta \leftarrow \emptyset$ 
2: while  $T_\pi \neq \emptyset$  do
3:    $t_\alpha \leftarrow \text{first}(T_\pi)$ 
4:   for all  $t_\gamma \in T - T_\Delta - \{t_\alpha\}$  do
5:     if  $\text{inconsistent}(t_\alpha \wedge \neg t_\gamma)$  then
6:        $T_\Delta \leftarrow T_\Delta \cup \{t_\gamma\}; T_\pi \leftarrow T_\pi - \{t_\gamma\}$ 
7:     end if
8:   end for
9:    $T_\pi \leftarrow T_\pi - \{t_\alpha\}$ 
10: end while
11: return  $T - T_\Delta$ 

```

Table 6.2: Example trace of WIPEOUTR_T with $T = \{t_1 : ABtesting = t, t_2 : statistics = f, t_3 : ABtesting = t \wedge statistics = t\}$. The result is the set $T - T_\Delta = \{t_1 \dots t_3\} - \{t_1\} = \{t_2, t_3\}$.

Iteration	T_π	T_Δ	t_α	t_γ
1.1	$\{t_1 \dots t_3\}$	$\emptyset$	t_1	t_2
1.2	$\{t_1 \dots t_3\}$	$\emptyset$	t_1	t_3
2.1	$\{t_2, t_3\}$	$\emptyset$	t_2	t_1
2.2	$\{t_2, t_3\}$	$\emptyset$	t_2	t_3
3.1	$\{t_3\}$	$\{t_1\}$	t_3	t_1
3.2	$\{t_3\}$	$\{t_1\}$	t_3	t_2

6.4 Redundancy Detection

Test Suite Redundancy Detection. WIPEOUTR_T (Algorithm 6.1) determines a maximal set of test cases that can be deleted from a test suite T guaranteeing that $\neg \exists \{t_i, t_j\} \subseteq T (i \neq j) : t_i \models t_j$. In this context, the mentioned deletion set is maximal if no test case remains in T that is redundant w.r.t. to the mentioned property. This means that all tests of the original (potentially redundant) test suite are taken into account (from the logical point of view) by the redundancy-free test suite.

WIPEOUTR_T is activated with a potentially redundant test suite T . In this context, all test cases t_α of T (represented by T_π) are checked with regard to potentially redundant test cases t_γ in $T - T_\Delta - \{t_\alpha\}$ where T_Δ contains already detected redundant test cases.

The redundancy check between t_γ and t_α can be performed on the basis of the consistency check (performed by a constraint solver) $\text{inconsistent}(t_\alpha \wedge \neg t_\gamma)$. If this check is fulfilled, this means that there does not exist a solution for $t_\alpha \wedge \neg t_\gamma$, i.e., there is no solution for t_α which is not already covered by t_γ (t_γ is redundant). All identified redundant test cases are stored in T_Δ . After having completed the redundancy checks for all test cases in T , WIPEOUTR_T(T) returns the redundancy-free test suite $T - T_\Delta$. An example execution trace of WIPEOUTR_T($T = \{t_1 \dots t_3\}$) is depicted in Table 6.2.

Algorithm 6.2 $\text{WIPEOUTR}_{FM}(CF = \{c_1 \dots c_n\})$

```

1:  $CF_{\Delta} \leftarrow CF$ 
2: for all  $c_{\alpha} \in CF$  do
3:   if  $\text{inconsistent}(CF_{\Delta} - \{c_{\alpha}\} \cup \{\neg c_{\alpha}\})$  then
4:      $CF_{\Delta} \leftarrow CF_{\Delta} - c_{\alpha}$ 
5:   end if
6: end for
7: return  $CF_{\Delta}$ 

```

Table 6.3: Trace of WIPEOUTR_{FM} with $CF = \{c_1 \dots c_8\}$ resulting in the redundancy-free feature model $CF = \{c_1 \dots c_7\}$.

Iteration	CF_{Δ}	c_{α}
1	$\{c_1 \dots c_8\}$	c_1
...	...	...
8	$\{c_1 \dots c_7\}$	c_8

WIPEOUTR_T Consistency Checks. In the worst case, the number of solver activations (consistency checks) of $\text{WIPEOUTR}_T(T)$ is $(n - 1) \times n$ (if no redundant constraints are contained in T) where $n = |T|$. In the best case, the number of needed consistency checks is $n - 1$ (if all constraints in T are redundant).

Feature Model Redundancy Detection. WIPEOUTR_{FM} (Algorithm 6.2) determines a maximal set of feature model constraints that can be deleted from CF guaranteeing that $\neg \exists c_{\alpha} \in CF : CF - \{c_{\alpha}\} \models c_{\alpha}$. The deletion set is maximal if no constraint remains in CF that is redundant w.r.t. to the mentioned property. This means that all non-redundant constraints of the original (potentially redundant) set CF are kept in CF in a semantic-preserving fashion, i.e., the solution space remains the same.

WIPEOUTR_{FM} is activated with a potentially redundant set of constraints CF . In this context, all constraints $c_{\alpha} \in CF$ are checked with regard to a potential redundancy of c_{α} . This is achieved by the test (solver call) $CF_{\Delta} - \{c_{\alpha}\} \cup \{\neg c_{\alpha}\}$. As long as $CF_{\Delta} - \{c_{\alpha}\}$ is inconsistent with $\{\neg c_{\alpha}\}$, the original solution space has not been changed. All identified non-redundant constraints of CF are stored in CF_{Δ} . After having completed the redundancy checks for all feature model constraints in CF , $\text{WIPEOUTR}_{FM}(CF)$ returns the redundancy-free set of domain constraints CF_{Δ} . An execution trace of $\text{WIPEOUTR}_{FM}(CF = \{c_1 \dots c_8\})$ is shown in Table 6.3.

WIPEOUTR_{FM} Consistency Checks. The number of consistency checks of WIPEOUTR_{FM} is $|CF|$ since for every constraint in CF , the corresponding redundancy property has to be checked.

Table 6.4: Characteristics of the Linux-2.6.33.3 feature model.

#features	6,467
#relationships	6,322
#cross-tree constraints	7,650
#mandatory	244
#optional	6,037
#alternative	39
#or	2
#requires	7,108
#excludes	205
#3 CNF	337

6.5 Performance Analysis

We have evaluated our algorithms using the *Linux-2.6.33.3* feature model (see Table 6.4) taken from Diverso Lab’s benchmark² (Heradio et al. 2022) and could detect a redundancy degree of 34.36% in the original feature model, i.e., compared to the 13,972 constraints in the original feature model, a corresponding redundancy-free feature model contains only 9,171 constraints.³ We analyzed different degrees of additionally induced redundancy ($\frac{|\text{redundant constraints}(CF)|}{|CF|}$) with regard to the impact on (1) runtime of WIPEOUTR_T and test case execution (Table 6.5) and (2) runtime of WIPEOUTR_{FM} and solution search on the basis of increased redundancy degrees in CF (Table 6.6). As a solver environment used for consistency checking in the discussed algorithms, we applied the CHOCO constraint solver.⁴ The results of our evaluation can be summarized as follows.

1. Regarding feature model testing operations, we could identify improvements in terms of reduced test case execution times in the case of non-redundant test sets (see Table 6.5). This result is relevant in the context of knowledge base (e.g., configuration knowledge bases or feature models) testing scenarios (Felfernig, Friedrich, Jannach, et al. 2004; Le, Felfernig, Tran, Atas, et al. 2021), where the number of needed test executions directly influences the performance of the underlying diagnosis algorithms and regression testing.
2. Regarding runtime reductions in solution search, we could also observe some performance improvements (see Table 6.6). These results have the potential to further improve solver performance in the context of feature model configuration scenarios. However, specifically in this context, a more detailed related analysis with further real-world feature models

²<https://github.com/diverso-lab/benchmarking>

³The dataset, the implementation of algorithms, and evaluation programs can be obtained from <https://github.com/AIG-ist-tugraz/WipeOutR>.

⁴<https://choco-solver.org>

Table 6.5: Avg. runtime (sec) of WIPEOUTR_T (each entry denotes WIPEOUTR_T runtime/execution runtime with redundant T/execution runtime with non-redundant T) evaluated on an Intel Core i7 (6 cores) 2.60GHz (16 GB of RAM).

T	redundancy (red.%)		
	0%	50%	90%
10	54 / 9.2 / 9.2	32 / 4.4 / 2.5	5 / 1.9 / 0.03
50	1,950 / 54 / 54	1,302 / 37 / 25	190 / 36 / 3
100	7,169 / 94 / 94	8,539 / 122 / 46	3,278 / 94 / 7.2
250	50,274 / 234 / 234	48,013 / 325 / 129	16,080 / 262 / 24

Table 6.6: Avg. runtime (in sec) of WIPEOUTR_{FM}(*) evaluated on an Intel Core i7 (6 cores) 2.60GHz (16 GB of RAM).

CF	red.%	runtime*	solv. (red.)	solv. (non-red.)
13,972	34.36%	1,846.04	0.73	0.63
18,342	50%	1,881.92	0.74	0.63
30,572	70%	1,778.25	0.74	0.60

is needed and in the scope of our future work. Our goal in this context is to understand in more detail which types of knowledge structures redundancy-freeness can help to significantly improve the performance of solution search.

6.6 Conclusions and Future Work

We have introduced algorithms for the detection of redundancies in feature models (FM) and FM test suites. Thus, we are able to support engineers in feature model development and are also able to potentially reduce configurator runtimes. A *threat to the validity* of our work is a still-needed in-depth analysis of further existing variability models with regard to their redundancy degree. In this chapter, we focused on the algorithmic level – explaining redundancies on the user interface level is an important issue for future work. *Further issues for future work* include (1) more in-depth evaluations on the basis of industrial feature models (in this context, we plan to analyze the different sources of redundancies in more detail), and (2) algorithmic improvements on the basis of a parallelization of the needed consistency checks (Vidal et al. 2021) (since redundancy detection can be heavily parallelized, such approaches have the potential to further improve the performance of redundancy detection).

Chapter 7

Test Case Aggregation for Efficient Feature Model Testing

This chapter is based on the results documented in (Le, Felfernig, and Tran 2022). The author of this thesis carried out the literature research, algorithm design, implementation, empirical evaluation, and data analysis, and contributed to the writing of the paper.

The increasing size and complexity of feature models (FM) requires the provision of efficient testing and debugging techniques. Feature models can be tested, for example, with regard to their conformance with a pre-defined set of analysis operations. In this chapter, we show how the number of consistency checks for FM testing can be reduced on the basis of test case aggregation.¹ Using a divide-and-conquer based approach, we show how to transform a feature model test suite into a corresponding aggregated representation where individual test cases can be combined if specific consistency criteria are fulfilled. Performance improvements are also analyzed on the basis of a best- and worst-case runtime analysis.

7.1 Introduction

Feature models (FM) are used to specify variability properties of complex items such as products, services, and software artefacts (Kang et al. 1990; Batory 2005; Apel and Kästner 2009; Benavides, Segura, and Ruiz-Cortés 2010; Acher et al. 2018). The development of feature models is often an error-prone process where aspects such as limited domain knowledge, cognitive overloads, and outdated constraints can lead to erroneous models not reflecting the variability and commonality properties of the underlying item (application) domain, for example, configurations that should be supported are not supported and – vice-versa – configurations that are supported by the feature model should not be supported. Erroneous FMs can be characterized by a subset of constraints

¹The work presented in this chapter has been developed within the scope of the OPENSPACE project, which is funded by the Austrian Research Promotion Agency (FO999891127).

responsible for faulty (unintended) model semantics (Batory 2005; Schobbens, Heymans, and Trigaux 2006). Such models can result in FM configurations which do not exist in the corresponding item (application) domain.

As a consequence, FM development must be supported by automated testing and debugging techniques (Le, Lee, et al. 2013; Le, Felfernig, Uta, Benavides, et al. 2021). Feature model testing can be supported, for example, by *analysis operations* that help to assure the conformance of feature models with a predefined set of well-formedness rules (Mendonca, Wąsowski, and Czarnecki 2009; Benavides, Segura, and Ruiz-Cortés 2010). Such well-formedness rules can be regarded as *domain-independent* test cases that help to assure reasonable properties expected to be fulfilled by feature models. An example of such a property is the *non-existence of dead features*, i.e., features which are inactive in every possible feature model configuration. Test cases can also be *domain-specific* (Le, Felfernig, Uta, Benavides, et al. 2021), i.e., can define expected domain-specific variability properties to be fulfilled by every feature model configuration. Feature model testing can be combined with corresponding diagnosis operations. The approach behind is to exploit test cases for the induction of *conflicts* (Junker 2004) in feature models (some of the test cases are inconsistent with the feature model constraints), which can then be resolved on the basis of the concepts of model-based diagnosis (Reiter 1987; Felfernig, Schubert, and Zehentner 2012; Hentze et al. 2021). In this context, test cases are typically part of test suites which are used to specify the intended semantics of a feature model (Le, Felfernig, Uta, Benavides, et al. 2021).

Checking the consistency between feature model test cases and a feature model (FM) can be time-consuming since individual test case checking requires a constraint or SAT solver (Rossi, Beek, and Walsh 2006). Consistency checking means that, for example, a constraint solver has to figure out if a test case allows the calculation of a solution when included as an additional constraint into the FM. Such testing operations can trigger *scalability issues* (Fraser and Wotawa 2007) for FM testing as well as the diagnosis of FMs in situations where some of the test cases become inconsistent (Le, Felfernig, Uta, Benavides, et al. 2021). In this chapter, we present a first approach to the *aggregation of individual test cases* that can help to significantly reduce the overall amount of consistency checks needed in testing operations. Thus, we show how the efficiency of feature model testing can be improved by reducing the number of consistency checks needed to check each individual test case part of a feature model test suite. In this context, we introduce the concept of *aggregated test cases* and show how such test cases can be derived from a given test suite.

The remainder of this chapter is organized as follows. In Section 7.2, we introduce an example *survey software* feature model. Thereafter, in Section 7.3 we show basic approaches to the testing of feature models. Our approach to improve the performance of feature model testing on the basis of *aggregated test cases* is introduced in Section 7.4. In Section 7.5, we discuss threats to validity and open issues for future work. The chapter is concluded with Section 7.6.

7.2 Example Feature Model

Features are organized hierarchically (see Figure 7.1) using relationships such as *mandatory* (e.g., each *survey* software includes a *license* model), *optional* (e.g.,

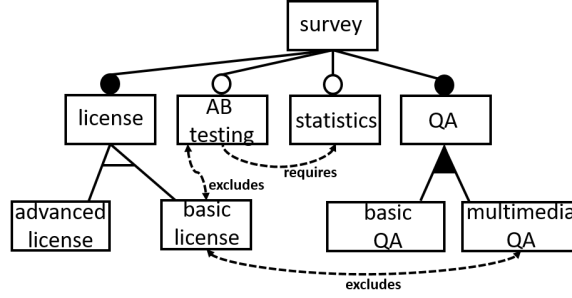


Figure 7.1: An example feature model (*survey software*) (Le, Felfernig, Uta, Benavides, et al. 2021).

a *statistics* feature could be included), *alternative* (e.g., a *license* is either and *advanced* or a *basic* license), and *or* (e.g., questions can be defined as *basic* or as *multimedia* ones). In addition to hierarchical relationships, variability properties can also be defined on the basis of so-called cross-tree constraints in terms of *excludes* (e.g., the *basic* license model does not allow the inclusion of *ABtesting*) and *requires* relationships (e.g., the inclusion of the *ABtesting* feature requires the inclusion of *statistics*). To support reasoning about feature model properties, feature models have to be translated into a corresponding formal representation, for example, a feature model can be represented as a configuration task (Benavides, Trinidad, and Ruiz-Cortez 2005) – such a definition on the basis of a constraint satisfaction problem (CSP) is the following (Definition 7.1).

Definition 7.1 (FM Configuration Task). A feature model (FM) *configuration task* can be defined as a constraint satisfaction problem (CSP) (F, D, C) where $F = \{f_1, f_2 \dots f_n\}$ is a set of variables, $D = \{dom(f_1), dom(f_2) \dots dom(f_n)\}$ is a set of variable domain definitions where $dom(f_i) = \{1, 0\}$, and $C = C_R \cup C_F$ where $C_R = \{c_1, c_2 \dots c_k\}$ is a set of customer requirements (i.e., preferred inclusions and exclusions of features) and $C_F = \{c_{k+1}, c_{k+2} \dots c_q\}$ is a corresponding set of constraints derived from the feature model.

A set of rules that define how to translate a feature model into a corresponding constraint-based representation is discussed a.o. in Benavides et al. (Benavides, Segura, and Ruiz-Cortés 2010). Table 7.1 depicts the constraints $c_i \in C_F$ that have been derived from the feature model shown in Figure 7.1.

Given the definition of a feature model configuration task, we are able to introduce the definition of a corresponding feature model configuration (see Definition 7.2).

Definition 7.2 (FM Configuration). A feature model (FM) *configuration* for an FM configuration task (F, D, C) is an assignment A of the variables in F which fulfills the criteria that all constraints in C are consistent with the variable assignments in A .

An example FM configuration² is $A = \{survey = 1, license = 1, ABtesting = 0, statistics = 1, QA = 1, advancedlicense = 0, basiclicense = 1, basicQA = 1, multimediaQA = 0\}$.

²In this chapter, we use $f_i = 1$, $f_i = true$, and f_i synonymously.

Table 7.1: Feature model constraints $C_F = \{c_1 \dots c_9\}$.

constraint	CSP representation
c_0	$survey = 1$
c_1	$survey \leftrightarrow license$
c_2	$ABtesting \rightarrow survey$
c_3	$statistics \rightarrow survey$
c_4	$survey \leftrightarrow QA$
c_5	$QA \leftrightarrow basicQA \vee multimediaQA$
c_6	$(advancedlicense \leftrightarrow \neg basiclicense \wedge license) \wedge$ $(basiclicense \leftrightarrow \neg advancedlicense \wedge license)$
c_7	$\neg(ABtesting \wedge basiclicense)$
c_8	$ABtesting \rightarrow statistics$
c_9	$\neg(basiclicense \wedge multimediaQA)$

7.3 Feature Model Testing

As already mentioned, feature models can be tested on the basis of test cases (Le, Felfernig, Uta, Benavides, et al. 2021). Test cases (see Definition 7.3) can represent basic feature model analysis operations (Benavides, Segura, and Ruiz-Cortés 2010) such as void features or domain-specific properties, for example, *there should exist at least one possible configuration* that includes the features *basiclicense* and *statistics*. This would result in a corresponding test case $basiclicense \wedge statistics$. Consequently, in our approach the notion of a test case is simple: *a test case is interpreted as a constraint that exemplifies the intended semantics of a feature model*.

Feature Model Analysis Operations as Test Cases. Test cases can also be used to represent different properties required by feature model analysis operations (but are not limited to such types of feature model properties). For example, *void features* are associated with an analysis operation (Benavides, Segura, and Ruiz-Cortés 2010) that tries to detect such features which should not be part of a feature model. A corresponding feature model test case that helps to assure that a feature f_i isn't of type *void* is $f_i = true$, i.e., if the test case can not be fulfilled, the corresponding feature is of type *void*. Similarity, we can define a test case $f_{root} = true$ (f_{root} is the root feature of the feature model) which assures that there exists at least one configuration for a given feature model. Finally, a test case $f_i = false$ can be used to specify that a feature f_i should not be of type *false optional*.

The sources of test cases can be quite diverse. Test cases can be generated from different analysis operations (e.g., void features), from already completed consistent feature model configurations, or can be manually defined by knowledge engineers within the scope of feature model development and maintenance.

Definition 7.3 (FM Test Case). In constraint-based feature model (FM) representations, an FM test case can be defined as a constraint t that represents a specific intended semantics of a feature model.

Following this definition, $t_1 : basiclicense \wedge statistics$ can be regarded as an

example of a feature model test case. A combination of individual test cases is denoted as test suite (see Definition 7.4).

Definition 7.4 (FM Test Suite). When using constraint-based feature model representations, a test suite $T = \{t_1 \dots t_k\}$ where t_i represents individual FM test cases. A feature model is consistent with regard to a given test suite T , if $\forall t_i \in T : \text{consistent}(t_i \cup C_F)$.

Example Test Suite T . Following this definition, we can regard $T = \{t_1 : \text{basiclicense} \wedge \text{statistics}, t_2 : \text{basiclicense} \wedge \text{basicQA}, t_3 : \text{ABtesting} \wedge \text{multimediaQA}\}$ as a simple example of an FM test suite. The test suite T is consistent with the constraints C_F of our example feature model (see Table 7.1) since $\text{consistent}(\{t_1\} \cup C_F)$, $\text{consistent}(\{t_2\} \cup C_F)$, and $\text{consistent}(\{t_3\} \cup C_F)$.

Existing feature model regression testing and diagnosis approaches follow the idea of testing each $t_i \in T$ for consistency with regard to the constraints C_F part of the feature model (Le, Felfernig, Uta, Benavides, et al. 2021) (see Definition 7.4). If we take a more detailed look at the test cases in T , we can see that the test cases t_1 and t_2 must not necessarily be checked individually but can be transformed into the aggregated test case $T_\alpha = \{t_1, t_2\}$ which represents the constraint (test case) $\text{basiclicense} \wedge \text{statistics} \wedge \text{basicQA}$. If we rely on the assumption of local changes in feature model development and maintenance, i.e., not every extension or adaptation has an impact on the whole feature model and solution space, we can expect that the consistency of the majority of test cases remains stable, i.e., does not become outdated from one maintenance step to the next.

If a feature model gets adapted, it can be the case that some test cases become inconsistent with the new version of the feature model. If we assume that the test suites contain only valid test cases, i.e., test cases that reflect examples of settings existing in the underlying item (application) domain, some of the constraints in the feature model can be regarded erroneous. If, however, an *aggregated test case* T_α becomes inconsistent with C_F , there is a need to restore the individual test cases of T_α and to perform consistency checks on the level of the individual test cases. For example, if a knowledge engineer includes an additional constraint $c_{10} : \neg(\text{statistics} \wedge \text{basicQA})$ into the feature model and excludes the constraint c_9 (i.e., $C_F \leftarrow C_F \cup \{c_{10}\} - \{c_9\}$), then our example $T_\alpha = \{t_1, t_2\}$ becomes inconsistent with C_F , i.e., $\{t_1 \cup t_2\} \cup C_F$ is inconsistent. However, C_F is still consistent with the individual test cases t_1 , t_2 , and t_3 . In other words, the constraints in C_F fulfill each $t_i \in T$. Consequently, aggregated tests T_α can help to improve the performance of testing operations. Aggregates T_α help reduce the number of needed consistency checks when performing FM testing but also when performing FM diagnosis which follows the goal of restoring the consistency between feature model and test suite (Le, Felfernig, Uta, Benavides, et al. 2021).

7.4 Determining Aggregated Test Cases

Following the idea of *aggregated test cases* introduced in the previous section, we introduce an algorithm that can be used to derive aggregated test cases T_α out from a test suite T .

Algorithm 7.1 AGGREGATEDTEST($T = \{t_1 \dots t_k\}, C_F$): T_α

```

1: if ISCONSISTENT( $\{\bigwedge_{t_i \in T}\} \cup C_F$ ) then
2:   return  $T$ 
3: end if
4: if  $|T| = 1$  then
5:   return  $\emptyset$ 
6: end if
7:  $l = \lfloor \frac{l}{2} \rfloor$ 
8:  $T_1 \leftarrow t_1 \dots t_l$ ;  $T_2 \leftarrow t_{l+1} \dots t_k$ ;
9:  $\theta_2 \leftarrow \text{AGGREGATEDTEST}(T_1, C_F)$ ;
10:  $\theta_1 \leftarrow \text{AGGREGATEDTEST}(T_2, C_F \cup \theta_2)$ ;
11: return  $\theta_1 \cup \theta_2$ 

```

Algorithm 7.1 is based on the idea of determining *one maximum aggregated test case at a time*. *Maximality of T_α* in this context denotes the fact that is not possible to add further test cases to T_α without violating the criteria of *consistent*($T_\alpha \cup C_F$). Consequently, determining a complete set of aggregated test cases for T requires an repeated activation of AGGREGATEDTEST($T - T_\alpha, C_F$) until $T - T_\alpha = \emptyset$. In this context, T_α always denotes the most recent aggregated test case returned by AGGREGATEDTEST.

Note that test cases of T_α are deleted from the test suite T since *individual aggregated test cases are assumed to be disjoint*. The AGGREGATEDTEST approach follows the concept of the FASTDIAG algorithm that can be used for determining maximal satisfying subsets (MSS) of constraint sets (Felfernig, Schubert, and Zehentner 2012) (see Definition 7.5).

Definition 7.5 (Aggregated Test). In the context of constraint-based feature model representations, an *aggregated test (case)* is a set $T_\alpha = \{t_1 \dots t_u\} \subseteq T$: *consistent*($\{t_1 \wedge \dots \wedge t_u\} \cup C_F$).

The set of all aggregated tests is represented by a corresponding *aggregated test suite* (see Definition 7.6). In this context, we require that in an aggregated test suite TA each test case t_i is assigned to *exactly one* aggregated test $T_{\alpha l} \in TA$, i.e., the assignment of test cases to aggregated tests has to be *disjoint*.

Definition 7.6 (Aggregated Test Suite). When using constraint-based feature model representations, an aggregated test suite is a set $TA = \{T_{\alpha 1} \dots T_{\alpha u}\} : t_i \in T \implies \exists l : t_i \in T_{\alpha l}$ (*disjoint property*).

In our example setting, $\{t_1 \wedge t_2 \wedge t_3\} \cup C_F$ is *inconsistent*. In this situation, AGGREGATEDTEST will divide T into the two subsets $T_1 = \{t_1\}$ and $T_2 = \{t_2, t_3\}$. In the following, t_1 is regarded as element of T_α and the algorithm checks if $T = t_1 \wedge t_2$ is consistent with C_F (ISCONSISTENT($\{t_1 \wedge t_2\} \cup C_F$)). Since this is the case and t_3 is the last remaining test case, t_3 is removed (taking into account that $T \cup C_F$ is inconsistent). Consequently, $T_{\alpha 1} = \{t_1, t_2\}$ is returned as answer to the call AGGREGATEDTEST(T, C_F). The next activation is AGGREGATEDTEST($T - \{t_1, t_2\}, C_F$) resulting in $T_{\alpha 2} = \{t_3\}$. Thereafter, no further activation of AGGREGATEDTEST is needed since $T - \{t_1, t_2\} - \{t_3\} = \emptyset$. The *aggregated test suite* generated on the basis of AGGREGATEDTEST is $TA = \{T_{\alpha 1} : \{t_1, t_2\}, T_{\alpha 2} : \{t_3\}\}$.

Reduction of Consistency Checks. If the conjunction of all test cases in T is consistent with C_F , AGGREGATEDTEST would return $T_\alpha = T$. Assuming $|T| = r$, the originally needed r consistency checks would be replaced by the need of *only one consistency check* for T_α , i.e., $\text{consistent}(\{t_1 \wedge \dots \wedge t_r\} \cup C_F)$. In our example, three consistency checks are needed for checking the consistency of each $t_i \in T$ with C_F . Only two checks are needed with the aggregated tests $T_{\alpha 1} = \{t_1, t_2\}$ and $T_{\alpha 2} = \{t_3\}$, namely (1) $\text{consistent}(\{t_1 \wedge t_2\} \cup C_F)$ and (2) $\text{consistent}(\{t_3\} \cup C_F)$. The worst case in terms of $\#$ needed consistency checks is reflected by $\forall_{\{t_i, t_j\} \subseteq T(i < j)} \text{inconsistent}(\{t_i \wedge t_j\} \cup C_F)$, i.e., for the given test suite T , no further aggregations are possible, i.e., $|T| = |TA|$.

Handling Inconsistent Aggregated Test Cases. Due to changes in the feature model, aggregated test cases can become inconsistent. This does not necessarily mean that the individual test cases in T_α are inconsistent, however, there does not exist a solution for the union of all individual test cases. When being confronted with such a situation in feature model regression testing or feature model diagnosis (Le, Felfernig, Uta, Benavides, et al. 2021), T_α has to be split and the resulting test cases have to be checked individually.

Evaluation Settings. For evaluation purposes, we (1) created synthesized feature models (see Table 7.2) using the Betty framework (Segura et al. 2012). As parameters, we chose (1) number of constraints in the feature model = $|C_F|$ with a ratio of 30% cross-tree constraints and (2) the number of features equals the number of constraints $\times 0.75$. The real-world feature models (see Table 7.3) have been selected from the SPLOT repository³ depending on the number of constraints ($|C_F|$). For each setting, we generated a test suite with four types of test cases: dead features, false optional, full mandatory, and partial configurations (including randomized feature settings).

Evaluation of Solver Performance. Tables 7.2–7.3 depict the results of an initial performance analysis⁴ we have conducted with the CHOCO constraint solving environment.⁵ We have evaluated the runtime of the solver (3 repetitions per setting) depending on the degree of reduction of needed consistency checks for a test suite T . For example, in Table 7.2, for a constraint set ($|C_F| = 10$) with $|T| = 60$ test cases, the runtime for completing all (individual) consistency checks is 1.10 msec. The runtime needed for completing the consistency checks for an aggregated test suite $|TA| = 5$ is 0.53 msec.

Usage of AGGREGATEDTEST. The algorithm is intended to be applied offline, i.e., if new test cases have been specified or previous test cases have been adapted or deleted, AGGREGATEDTEST must not necessarily be re-activated – this can be done in batch-style since changes can be handled as follows: new test cases can simply be added to the test suite, deleted test cases can be removed from the test suite, and changed test cases can assumed to be deleted and re-inserted to the test suite as new non-aggregated test case. At pre-defined points of time, AGGREGATEDTEST can be applied to generate a new version of the

³<http://www.splot-research.org/>

⁴The experiments were conducted on an Intel Core i7 (6 cores) 2.60GHz, 16GB RAM.

⁵choco-solver.org.

7.5. Threats to Validity & Open Issues

Table 7.2: Avg. runtime (in *msec*) of CHOCO depending on the reduction of needed consistency checks for T for synthesized constraint sets C_F using the BETTY generator (Segura et al. 2012).

$ C_F $	$ T =60$	$ TA =30$	$ TA =15$	$ TA =10$	$ TA =5$
10	1.10	1.06	0.98	0.85	0.53
25	1.22	1.04	0.80	0.80	0.40
50	2.42	1.60	1.33	1.11	0.75
100	4.85	2.61	1.41	1.10	0.86
500	15.32	9.07	6.32	6.42	3.29
1000	15.98	10.77	5.21	5.14	4.83

Table 7.3: Avg. runtime (in *msec*) of CHOCO depending on the reduction of needed consistency checks for T for constraint sets C_F selected from S.P.L.O.T. (*splot-research.org*).

$ C_F $	$ T =60$	$ TA =30$	$ TA =15$	$ TA =10$	$ TA =5$
10	0.96	0.81	0.76	0.68	0.39
25	1.89	1.32	0.95	0.88	0.69
50	3.82	2.29	1.38	1.26	0.76
100	5.62	3.10	2.01	1.58	1.00
500	32.70	22.59	11.12	9.02	5.17
1000	81.80	42.03	20.37	19.00	9.41

test suite. Following this strategy, the AGGREGATEDTEST execution time does not need to be taken into account when analyzing the performance of test case execution.

7.5 Threats to Validity & Open Issues

We have shown potential runtime improvements that can be achieved by aggregated tests. More in-depth performance evaluations with complex industrial feature models are planned. One assumption of our work is that test suites remain consistent. However, in real-world scenarios, test cases can become outdated. This aspect has to be taken into account in FM testing and debugging. The presented work is limited to the analysis of feature model testing scenarios. However, test suites also play a major role in feature model debugging. An analysis of potential related performance improvements is an issue for future work. Finally, the used optimization criteria for determining individual aggregated test sets is maximality, i.e., it is impossible to extend T_α with further constraints without making the conjunction of constraints in T_α inconsistent with C_F . Alternative approaches, for example, using hitting set directed acyclic graphs (HSDAG) (Reiter 1987) would return different maximal satisfiable subsets. A comparison between HSDAG based approaches and our approach is a topic of future work.

7.6 Conclusions

We have introduced an approach to increase the performance of feature model testing. This approach is based on a divide-and-conquer based strategy to the determination of *aggregated test cases*. On the logical level, aggregated test cases represent conjunctions of individual test cases. The underlying assumption is that test case aggregation helps to reduce the number of needed constraint solver activations. We have shown the applicability of our approach on the basis of a first performance evaluation.

Chapter 8

Conclusions and Future Work

This thesis has investigated *diagnostic reasoning* in *Constraint-based systems (CBSs)*, with a focus on *configuration knowledge bases (CKBs)* and *Feature Models (FMs)*. Across the design, maintenance, and usage phases of the model lifecycle, diagnostic reasoning supports the resolution of usage-time inconsistencies and the proactive *quality assurance (QA)* of models at design and maintenance time. The aim of the thesis has been to make diagnostic reasoning more efficient, more robust, and scalable on solver-agnostic foundations that interact with the underlying solver only through consistency checks (Gupta, Genc, and O’Sullivan 2021). The six contributions are organized around two pillars. The first pillar, *Efficient Conflict Detection and Diagnosis* (RQ1–RQ3), advances the core algorithms for identifying conflicts and computing diagnoses. The second pillar, *Diagnosis-inspired Quality Assurance for Feature Models* (RQ4–RQ6), extends diagnostic reasoning to QA tasks in feature model engineering. In this chapter, we summarize the contributions as answers to the research questions of Section 1.2, discuss the limitations of our approaches, and outline directions for future work.

8.1 Conclusions

RQ1 – Robust LCD (Chapter 2). *How can lazy conflict detection be made more robust and efficient through multi-conflict extraction and diversity control mechanisms?* *Lazy Conflict Detection (LCD)* precomputes minimal conflict sets offline through a genetic search over candidate constraint subsets (Uran and Felfernig 2018), but its baseline implementation suffers from low coverage, premature convergence, and unstable results across runs. In Chapter 2, we proposed Robust LCD. The algorithm integrates *Hitting Set Directed Acyclic Graph (HSDAG)* (Reiter 1987) with QUICKXPLAIN (Junker 2004) to extract multiple minimal conflict sets from each inconsistent individual. It also adds three diversity control mechanisms: full population reproduction, weighted genetic operators that bias selection toward conflict-rich individuals, and adaptive extinction that resets stagnant populations. On five real-world CKBs, Robust LCD recovers up to 85% of known conflict sets, reduces solver calls by up to 73%, and substantially improves result stability over the baseline. A nine-configuration ablation further shows that the four mechanisms play complemen-

tary roles. Multi-conflict extraction drives coverage, adaptive extinction reduces solver effort, and the diversity-control mechanisms stabilize the search across runs. Combinations that pair multi-conflict extraction with adaptive extinction sit on the Pareto frontier of coverage against solver effort across all five CKBs, while the best overall configuration varies per CKB. Convergence curves further show that adaptive extinction (alone or combined with the other mechanisms) accumulates conflicts rapidly in early generations while the baseline stagnates almost immediately, and that multi-conflict extraction alone trades early-stage speed for higher asymptotic coverage (see Section 2.5). The orchestration of these mechanisms remains static across application domains, motivating an investigation of adaptive activation under search-dynamics signals.

RQ2 – INFORMEDQX (Chapter 3). *How can knowledge from previously detected conflicts be systematically reused to improve the efficiency of conflict detection?* The QUICKXPLAIN algorithm (Junker 2004) treats each activation as an independent computation that repeats consistency checks across iterative configuration sessions in which similar constraint sets recur. In Chapter 3, we proposed INFORMEDQX, a reuse-aware variant of QUICKXPLAIN. The algorithm stores previously identified conflict sets in a lookup table (compressed as a *Binary Decision Diagram (BDD)*) and prunes the search space whenever a stored conflict is still present in the current constraint set. A bitmap-based preference function ranks reusable conflicts by user-defined priorities. On the *Linux-2.6.33.3* CKB, INFORMEDQX achieves runtime reductions of up to 45% and full elimination of solver calls when the preferred conflict can be retrieved from the lookup table. Under such perfect-reuse conditions, the avoided solver calls increase with $|CS|$, so the absolute savings of INFORMEDQX scale up with conflict size. When the stored conflict is only partially preferred, pruning of the consideration set yields moderate runtime reductions. When no stored conflict matches the current requirements, INFORMEDQX is marginally slower than QUICKXPLAIN, so reuse is most effective in iterative configuration sessions in which similar constraint sets recur (see Section 3.5). The reuse mechanism is currently bound to conflict detection, leaving its integration with direct diagnosis open.

RQ3 – FASTDIAGP (Chapter 4). *How can parallelization and speculative computation be integrated into direct diagnosis to reduce runtime on multi-core systems?* Direct diagnosis as realized by FASTDIAG (Felfernig, Schubert, and Zehentner 2012) executes its consistency checks sequentially, because each branching decision depends on the result of the previous check, so multi-core hardware remains underutilized. In Chapter 4, we proposed FASTDIAGP, which adopts the principle of speculative programming (Burton 1985) to pre-compute consistency checks along multiple possible execution paths before the actual path is known. A configurable look-ahead strategy distributes the speculative checks across processor cores. An empirical evaluation on the *Linux-2.6.33.3* CKB shows runtime reductions of up to 40% at 8 CPU cores, but the speedup curve saturates at this point and reverses beyond it. The speedup also decreases monotonically with diagnosis cardinality, so larger diagnostic problems benefit less from parallelism rather than more, and at $|\Delta| = 16$ scaling from 8 to 32 cores actively increases runtime. Capping the speculative depth at a fixed value,

independent of the number of cores, neutralizes this effect, so stable runtime across configurations confirms that speculative depth rather than core count is the controlling parameter. These findings outline an inherent ceiling for direct diagnosis. Speculation pre-computes consistency checks for multiple potential paths, but FD eventually follows only one of them, so the fraction of useful speculation cannot grow linearly with the number of cores (see Section 4.7). The look-ahead schedule is fixed at a single search strategy.

RQ4 – DIRECTDEBUG (Chapter 5). *How can direct diagnosis techniques be adapted to support automated testing and debugging of feature models?* Test-case-driven diagnosis was originally addressed by conflict-driven approaches (Felfernig, Friedrich, Jannach, et al. 2004) that first detect conflicts and then derive diagnoses through hitting-set construction (Reiter 1987), while direct diagnosis (Felfernig, Schubert, and Zehentner 2012) had not been adapted to integrate test suites at the meta-level. In Chapter 5, we proposed DIRECTDEBUG, which extends direct diagnosis to identify minimal sets of faulty constraints in a feature model with respect to a given set of test cases without predetermining conflicts. The integration is realized at the ISCONSISTENT step, where each activation triggers one consistency check per test case while the divide-and-conquer recursion of FD (Felfernig, Schubert, and Zehentner 2012) remains unchanged. The algorithm returns the *Maximal Satisfiable Subset* (MSS) Γ and derives the diagnosis as $\Delta = C \setminus \Gamma$, so it inherits the divide-and-conquer regime of FAST-DIAG without resorting to HSDAG-style conflict enumeration. On synthesized FMs from the Betty generator (Segura et al. 2012), DIRECTDEBUG runs in the millisecond range for moderate model sizes and remains tractable up to $|CF| = 1,000$ constraints with $|T| = 500$ test cases (under six seconds). The runtime scales roughly linearly with $|T|$ but super-linearly with $|CF|$, and a tenfold increase in $|CF|$ raises runtime by roughly $36\times$ at $|T| = 500$ (see Section 5.6), so constraint count drives runtime far more than test-suite size. The approach does not provide a method for generating test cases under coverage criteria, does not rank competing diagnoses by relevance, and does not explain its diagnoses at the user-interface level.

RQ5 – WIPEOUTR (Chapter 6). *How can redundant constraints in feature models and redundant test cases in test suites be automatically identified?* Prior approaches to redundancy detection either missed redundancies (Piette 2008; Sabin and Freuder 1999) or were tied to specific solver contexts such as linear programming (Paulraj and Sumathi 2010; Fukada, Gärtner, and Szedlak 2015), leaving feature model engineering without a complete, solver-independent framework. In Chapter 6, we proposed WIPEOUTR, which comprises two algorithms. WIPEOUTR_{FM} identifies a maximum set of constraints that can be removed from a feature model without changing its semantics. WIPEOUTR_T detects test cases in a test suite whose removal does not reduce fault detection capability. WIPEOUTR_{FM} requires $|CF|$ consistency checks (one per constraint), while WIPEOUTR_T performs up to $(n-1) \cdot n$ pairwise subsumption checks, an asymmetry that reflects the global vs pairwise nature of the two redundancy types. An empirical evaluation on the *Linux-2.6.33.3* feature model detects a redundancy degree of 34.36% (4,801 of 13,972 constraints), which suggests that redundancy is a systematic property of industrial FMs rather than an edge case.

Removing redundant test cases reduces test-suite execution time disproportionately, with up to a tenfold speedup at $|T| = 250$ under 90% redundancy. The WIPEOUTR_{FM} runtime itself stays essentially flat across redundancy degrees, so cost is predictable from $|CF|$ alone. The WIPEOUTR_T runtime instead grows roughly quadratically with $|T|$, while a 90% redundancy degree compresses this cost by up to threefold through earlier pruning (see Section 6.5). The redundancy checks are performed sequentially, although the underlying consistency checks are independent and parallelizable. Neither algorithm currently explains why a constraint or test case is found to be redundant.

RQ6 – AGGREGATEDTEST (Chapter 7). *How can test cases be aggregated to minimize the number of consistency checks required for feature model testing?* Industrial-scale regression validation re-executes large feature model test suites under a one-test-per-call schedule, which becomes a scalability bottleneck as FMs evolve (Fraser and Wotawa 2007; Mendonca, Wąsowski, and Czarnecki 2009). In Chapter 7, we proposed **AGGREGATEDTEST**, a divide-and-conquer method built on the MSS computation of **FASTDIAG** (Chapter 4) (Felfernig, Schubert, and Zehentner 2012). The method combines compatible test cases (test cases whose conjunction remains consistent with the model) into batches that are checked together in a single consistency check. The aggregation kernel reuses the divide-and-conquer regime of **FASTDIAG** (Felfernig, Schubert, and Zehentner 2012), where a maximum aggregated test T_α is the MSS of T with C_F as background knowledge. The optimization criterion is *maximality*. Each T_α cannot be extended further without violating consistency with C_F , while alternative **HSDAG**-based partitioning would yield different aggregations and remains open for comparison. On feature model evolution, an inconsistent aggregated test T_α is split back into individual test cases for downstream diagnosis, preserving compatibility with **DIRECTDEBUG**-style debugging (Chapter 5). The factor-12 reduction in solver activations (from $|T| = 60$ to $|TA| = 5$) translates into an 88% runtime reduction on the largest tested **SPLOT** model. The runtime gain is smaller than the activation gain because each aggregated check carries a larger conjunction. Real-world FMs from the **SPLOT** repository benefit more from aggregation than synthesized models of comparable size, where the runtime speedup at $|CF| = 1,000$ reaches $8.7\times$ versus $3.3\times$ on Betty-generated models (see Section 7.4). The aggregation is currently computed for a static test suite, and its behavior under suite evolution and incremental updates is unaddressed.

Limitations. Two thesis-level scope restrictions cut across the six contributions. First, the empirical evidence rests on a small set of canonical corpora. These include the five knowledge bases of Chapter 2, the *Linux-2.6.33.3* feature model used in Chapters 3, 4, and 6, the synthesized models from the Betty generator used in Chapters 5 and 7, and the real-world feature models from the **SPLOT** repository used in Chapter 7. Industrial FMs from other domains such as automotive or telecommunications remain out of scope. Second, the contributions assume a fixed constraint set during the diagnostic computation, so the online evolution of a knowledge base through new constraints, schema changes, or test-suite updates is not modeled. We address these scope restrictions in Section 8.2.

8.2 Future Work

For each contribution, we present two kinds of follow-up directions. *Thesis-level directions* span multiple chapters or extend the work over a longer horizon (cross-chapter synergies, longer-horizon extensions, and items addressing the scope restrictions of Section 8.1). *Chapter-specific directions* stay within the scope of the individual contribution and are drawn from that chapter’s own Future Work section. For each contribution, we give its thesis-level directions together with one or two of its most promising chapter-specific directions. The remaining items can be found in the Future Work section of each chapter.

Robust LCD. We will layer Monte Carlo tree search (Kocsis and Szepesvári 2006) on top of the genetic loop of Robust LCD as a local intensification of conflict-rich individuals. Each individual is a variable-value assignment, and the tree rooted at it explores neighbouring assignments that crossover and mutation may skip. The existing HSDAG-based extraction continues to harvest minimal conflicts at each inconsistent assignment the tree visits, and the tree’s reward counts the conflicts that are novel with respect to the repository. Stagnation of the tree then doubles as an additional trigger for adaptive extinction in the population. As a complementary chapter-specific direction, we will derive an adaptive activation policy that triggers the relevant diversity mechanism at runtime based on observable signals such as fitness stagnation or population entropy.

INFORMEDQX. We will explore neurosymbolic alternatives to the BDD-compressed lookup table of INFORMEDQX. *Logical Boltzmann Machines* (Tran and Garcez 2023) translate propositional formulae into the energy function of a restricted Boltzmann machine, with proven equivalence between logical satisfiability and energy minimization. Replacing the BDD by such a network would let INFORMEDQX retrieve the currently preferred conflict via a single free-energy minimization step on parallel neural hardware, in constant inference time with respect to the number of stored conflicts, and on a knowledge-representation-agnostic substrate that covers Answer Set Programming, Boolean Satisfiability, and description logics under the same architecture. The substitution also opens the door to *learning* preference patterns from historical conflict-detection sessions rather than relying on a fixed bitmap-based ranking. As a complementary chapter-specific direction, we will adapt the informed lookup mechanism to direct diagnosis by porting it to FASTDIAG (Felfernig, Schubert, and Zehentner 2012), extending the benefits of conflict reuse from QUICKXPLAIN to direct diagnosis under the same theoretical guarantees.

FASTDIAGP. We will study lower-overhead LOOKAHEAD implementations and adaptive speculation budgets, together with alternative parallelization paradigms that target the parallel decision logic of FASTDIAG itself rather than only the consistency-check generation. The aim is to remove the saturation observed at 8 cores of FASTDIAGP by re-architecting where parallelism is introduced in the algorithm. As a complementary chapter-specific direction, we will analyze alternative look-ahead search strategies, in particular breadth-first

variants and priority-driven schedules, with respect to their effect on speculative hit rates and overall runtime.

DIRECTDEBUG. We will introduce parallelism into DIRECTDEBUG along two complementary axes. First, the $|T|$ test-case-level consistency checks at each ISCONSISTENT invocation are independent and can be dispatched across multiple cores. Second, the divide-and-conquer recursion of DIRECTDEBUG can be parallelized through speculative computation as in FASTDIAGP (Chapter 4), pre-computing consistency checks along multiple paths before the actual path is chosen. The two axes are orthogonal, so a combined deployment compounds inner-loop parallelism with recursion-level speculation. As complementary chapter-specific directions, we will study the automated generation of feature model test cases under different coverage metrics so that the resulting test suite captures a wider spectrum of model behaviors, and use feature model quality metrics to predict which diagnosis is most relevant for a given inconsistency.

WIPEOUTR. The redundancy detection of WIPEOUTR can grow into a comprehensive feature-model cleaning workflow that integrates redundancy elimination with dead-feature removal, false-optional resolution, and other anomaly fixes already supported by the FMTTESTING plug-in (Burgstaller et al. 2023). Such a workflow would unify multiple QA tasks under a single pass and amortize solver activations across the toolchain. As complementary chapter-specific directions, we will parallelize the redundancy checks to reduce wall-clock cost on large feature models, and analyze common patterns of redundancy in industrial FMs and test suites so that maintenance recommendations can target the underlying causes rather than only the individual constraints or test cases.

AGGREGATEDTEST. AGGREGATEDTEST computes its aggregation under the assumption that test suites remain consistent across runs. In real-world settings, individual test cases may become outdated as the feature model evolves. AGGREGATEDTEST will need to detect such invalidated tests, repair them by aligning with the new model semantics, and re-aggregate around the residual updates. As a complementary chapter-specific direction, we will extend aggregation from feature model testing to feature model debugging, where the same divide-and-conquer batching could reduce solver activations during diagnosis.

Cross-Cutting Directions. Beyond the per-chapter directions above, five cross-cutting items stand out at thesis scope.

Explainability of diagnostic and QA outputs. Diagnostic and QA reasoning produces its results in formal notation, whether conflict sets, diagnoses, redundant constraints and test cases, or aggregated tests. While this format is sufficient for trained knowledge engineers, it remains inaccessible to domain experts and non-expert users. Three complementary explanation angles can bridge this gap, two of them based on *Large Language Models (LLMs)*. First, *UI-level explanation* highlights the features, relationships, and constraints that participate in each output directly within the feature model editor. Second, *LLM-based translation* uses LLMs to convert each output into natural-language

descriptions for knowledge engineers or non-expert users. Third, *LLM-based causal explanation* goes beyond translation, with LLMs articulating why each element causally contributes to the inconsistency, redundancy, or aggregation. In both LLM-based angles, a verification layer grounds each generated claim against the formal output to prevent hallucinations.

Neural-symbolic acceleration of diagnosis. The efficiency of QUICK-XPLAIN, FASTDIAG, and their reuse-aware variants depends on the order in which constraints are processed and on the cost of repeated solver calls. Future research will study neural-symbolic acceleration along two complementary axes. First, *learned constraint orderings* derived from historical diagnosis sessions can guide the heuristic ordering used by INFORMEDQX and FASTDIAGP, replacing fixed prior orders with domain-adaptive policies. Second, neural models can *predict consistency* or *directly predict conflict sets and diagnoses*, with a symbolic verification layer that guarantees correctness on residual cases. Both axes aim to combine symbolic reasoning with statistical learning while preserving the soundness and completeness guarantees of the underlying diagnostic algorithms.

Online maintenance under feature model evolution. The thesis assumes a fixed constraint set during the diagnostic computation, while real-world FMs evolve continuously as features and cross-tree constraints are added or modified. We will develop incremental variants of DIRECTDEBUG and WIPEOUTR that update their results in response to local changes without full recomputation, addressing the static-knowledge-base scope restriction of Section 8.1. This direction is the natural extension of the test-case-driven diagnosis of Chapter 5 and the redundancy detection of Chapter 6 to the maintenance phase.

Generalization beyond feature models and tool support. Although FMs are the primary motivating domain of the thesis, the QA-relevant concerns the thesis addresses (inconsistencies, redundancies, and scalability bottlenecks) arise in many other CBSs. Examples include product configurators, recommender systems, and AI planning, where the proposed algorithms remain applicable through their solver-agnostic interface. We plan to adapt the algorithms to the constraint structures and solver interfaces of these domains, addressing the benchmark-dependence scope restriction of Section 8.1. As a complementary practical direction, the tool support accompanying the thesis (Section 1.5.2) provides a starting point for industrial adoption.

Constraint acquisition from examples. The MSS-based machinery underlying several contributions of this thesis targets the *diagnostic* problem of localizing faulty constraints in an existing knowledge base. The same machinery can be turned around to support constraint *acquisition*. Given a set of positive and negative examples classified as valid or invalid (by domain experts or system-generated oracles), maximum satisfiable subsets of the candidate-constraint bias yield candidate constraint theories that converge on the ground-truth model. A natural next step is to explore this acquisition setting as a counterpart to the diagnostic uses studied in this thesis.

As configurable systems grow in size and complexity, efficient and trustworthy diagnostic reasoning will be central to keeping them comprehensible for the

8.2. *Future Work*

engineers who maintain them and the users who depend on them.

Bibliography

- Acher, M. et al. (2018). “VARYLATEX: Learning Paper Variants That Meet Constraints”. In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS ’18. Madrid, Spain: Association for Computing Machinery, pp. 83–88.
- Apel, S. and C. Kästner (2009). “An Overview of Feature-Oriented Software Development”. In: *The Journal of Object Technology* 8.5, pp. 49–84.
- Apel, S., D. Batory, et al. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer.
- Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.
- Bakker, R. et al. (1993). “Diagnosing and solving over-determined constraint satisfaction problems”. In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, pp. 276–281.
- Baptiste, P. et al. (2006). “Constraint-Based Scheduling and Planning”. In: *Handbook of Constraint Programming*. Ed. by F. Rossi, P. van Beek, and T. Walsh. Amsterdam, The Netherlands: Elsevier, pp. 759–797.
- Batory, D. (2005). “Feature Models, Grammars, and Propositional Formulas”. In: *Software Product Lines Conference*. Vol. 3714. LNCS. Springer, pp. 7–20.
- Beek, M. ter et al. (2022). “Efficient static analysis and verification of featured transition systems”. In: *Empirical Software Engineering* 27.10.
- Benavides, D., P. Trinidad, and A. Ruiz-Cortez (2005). “Automated Reasoning on Feature Models”. In: *Proceedings of the International Conference on Advanced Information Systems Engineering*. Vol. 3520. LNCS. Porto, Portugal, pp. 491–503.
- Benavides, D., S. Segura, and A. Ruiz-Cortés (2010). “Automated Analysis of Feature Models 20 Years Later: A Literature Review”. In: *Information Systems* 35.6, pp. 615–636.
- Benavides, D., A. Felfernig, et al. (2013). “Automated Analysis in Feature Modelling and Product Configuration”. In: *Proceedings of the 13th International Conference on Software Reuse*. LNCS 7925. Pisa, Italy: Springer, pp. 160–175.
- Benavides, D., C. Sundermann, et al. (2025). “UVL: Feature modelling with the Universal Variability Language”. In: *J. Syst. Softw.* 225, p. 112326.
- Bendík, J. and I. Černá (2020). “MUST: Minimal Unsatisfiable Subsets Enumeration Tool”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, pp. 135–152.
- Benitez, F. S. et al. (2025). “UVL web-based editing and analysis with flamapy.ide”. In: *Proceedings of the 19th International Working Conference*

- on *Variability Modelling of Software-Intensive Systems*. VaMoS '25. Rennes, France: ACM, pp. 121–125.
- Bessiere, C. et al. (2013). “Constraint Acquisition via Partial Queries.” In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. Vol. 13, pp. 475–481.
- Bessiere, C. (2020). “Constraint Reasoning”. In: *A Guided Tour of Artificial Intelligence Research: Volume II: AI Algorithms*. Springer International Publishing, pp. 153–183.
- Bordeaux, L., Y. Hamadi, and H. Samulowitz (2009). “Experiments with Massively Parallel Constraint Solving”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. California, USA: Morgan Kaufmann, pp. 443–448.
- Brewka, G. (1989). “Preferred Subtheories: An Extended Logical Framework For Default Reasoning”. In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pp. 1043–1048.
- Burgstaller, T. et al. (2023). “FMTTESTING: A FEATUREIDE Plug-in for Automated Feature Model Analysis and Diagnosis”. In: *Proceedings of the 26th European Conference on Artificial Intelligence*. Vol. 372. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 3190–3195.
- Burton, F. W. (1985). “Speculative computation, parallelism, and functional programming”. In: *IEEE Transactions on Computers* C-34.12, pp. 1190–1193.
- Carneiro, D., P. Novais, and J. Neves (2013). “Using genetic algorithms to create solutions for conflict resolution”. In: *Neurocomputing* 109. New trends on Soft Computing Models in Industrial and Environmental Applications, pp. 16–26.
- Castillo, L. et al. (2005). *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice*. Vol. 117 of Frontiers in Artificial Intelligence and Applications. IOPress.
- Cheng, K. and R. Yap (2005). “Constrained Decision Diagrams”. In: *Proceedings of the 20th National Conference on Artificial Intelligence*. Vol. 1, pp. 366–371.
- Czarnecki, K., S. Helsen, and U. Eisenecker (2005). “Formalizing Cardinality-based Feature Models and their Specialization”. In: *Software Process: Improvement and Practice* 10.1, pp. 7–29.
- Erdem, E., M. Gelfond, and N. Leone (2016). “Applications of Answer Set Programming”. In: *AI Magazine* 37.3, pp. 53–68.
- Falkner, A. and G. Fleischanderl (2001). “Configuration requirements from railway interlocking stations”. In: *Proc. of the Configuration Workshop on 17th International Joint Conference on Artificial Intelligence (IJCAI-2001), Seattle, Washington (August 6 2001)*. Citeseer.
- Felfernig, A., G. Friedrich, D. Jannach, et al. (2004). “Consistency-based diagnosis of configuration knowledge bases”. In: *Artificial Intelligence* 152.2, pp. 213–234.
- Felfernig, A. and R. Burke (2008). “Constraint-Based Recommender Systems: Technologies and Research Issues”. In: *10th International Conference on Electronic Commerce*. ICEC '08. Innsbruck, Austria: Association for Computing Machinery.
- Felfernig, A., G. Friedrich, M. Schubert, et al. (2009). “Plausible Repairs for Inconsistent Requirements”. In: *Proceedings of the 21st International Joint*

- Conference on Artificial Intelligence*. California, USA: Morgan Kaufmann, pp. 791–796.
- Felfernig, A., M. Schubert, M. Mandl, et al. (2010). “Efficient Explanations for Inconsistent Constraint Sets”. In: *Proceedings of the 19th European Conference on Artificial Intelligence*. IOS Press, pp. 1043–1044.
- Felfernig, A., M. Schubert, and C. Zehentner (2012). “An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets”. In: *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 26.1, pp. 53–62.
- Felfernig, A., D. Benavides, et al. (2013). “Towards Anomaly Explanation in Feature Models”. In: *Proceedings of the 15th International Configuration Workshop*. Vienna, Austria, pp. 117–124.
- Felfernig, A., S. Reiterer, et al. (2014). “Conflict Detection and Diagnosis in Configuration”. English. In: *Knowledge-Based Configuration: From Research to Business Cases*. 1st ed. Netherlands: Elsevier B.V., pp. 73–87.
- Felfernig, A., L. Hotz, et al. (2014). *Knowledge-Based Configuration: From Research to Business Cases*. Morgan Kaufmann.
- Felfernig, A., R. Walter, et al. (2018). “Anytime Diagnosis for Reconfiguration”. In: *Journal of Intelligent Information Systems* 51.1, pp. 161–182.
- Felfernig, A., V.M. Le, and T.N.T. Tran (2020). “Supporting Feature Model-Based Configuration in Microsoft Excel”. In: 22nd International Configuration Workshop, (ConfWS’ 20), pp. 35–38.
- Felfernig, A., B. Ortner, and V.M. Le (2022). “Table-Based Knowledge Representations for Industrial Feature Models”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B. SPLC ’22*. Association for Computing Machinery, pp. 245–248.
- Felfernig, A., V.M. Le, and S. Lubos (2023). “Conjunctive Query Based Constraint Solving for Feature Model Configuration”. In: *Proceedings of the 12th Conference on Information Technology and Its Applications*. Vol. 734. Springer Nature Switzerland, pp. 357–367.
- Felfernig, A., A. Falkner, and D. Benavides (2024). “Feature Models: AI-Driven Design, Analysis and Applications”. In: *Feature Models: AI-Driven Design, Analysis and Applications*. Springer.
- Felfernig, A., D. Garber, et al. (2025). “Causality-based Explanations for Feature Model Configuration”. In: *Proceedings of the 19th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, pp. 86–90.
- Fleischanderl, G. et al. (1998). “Configuring Large Systems Using Generative Constraint Satisfaction”. In: *IEEE Intelligent Systems and their Applications* 13.4, pp. 59–68.
- Fleischanderl, G. (2002). “Suggestions from the Software Engineering Practice for Applying Consistency-Based Diagnosis to Configuration Knowledge Bases”. In: *13th Intl. Workshop on Principles of Diagnosis (DX-02)*. Semmering, Austria, pp. 33–35.
- Fraser, G. and F. Wotawa (2007). “Redundancy Based Test-Suite Reduction”. In: *10th International Conference on Fundamental Approaches to Software Engineering*. FASE’07. Braga, Portugal: Springer-Verlag, pp. 291–305.
- Freuder, E. C. and A. K. Mackworth (1994). *Constraint-based reasoning*. Vol. 58. 1-3. MIT press.

- Fukada, K., B. Gärtner, and M. Szedlak (2015). “Combinatorial Redundancy Detection”. In: *31st International Symposium on Computational Geometry*, pp. 315–328.
- Galindo, J.A., D. Benavides, and S. Segura (2010). “Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis”. In: *1st International Workshop on Automated Configuration and Tailoring of Applications*. Vol. 688. CEUR Workshop Proceedings. CEUR-WS.org, pp. 29–34.
- Galindo, J.A., D. Benavides, P. Trinidad, et al. (2019). “Automated Analysis of Feature Models: Quo Vadis?” In: *23rd International Systems and Software Product Line Conference - Volume A. SPLC '19*. Paris, France: ACM, p. 302.
- Galindo, J.A., J.-M. Horcas, et al. (2023). “FLAMA: A collaborative effort to build a new framework for the automated analysis of feature models”. In: *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B. SPLC '23*. Tokyo, Japan: ACM, pp. 16–19.
- Gent, I. et al. (2018). “A Review of Literature on Parallel Constraint Solving”. In: *Theory and Practice of Logic Programming* 18.5–6, pp. 725–758.
- Gregoire, E., B. Mazure, and C. Piette (2008). “On Finding Minimally Unsatisfiable Cores of CSPs”. In: *International Journal on Artificial Intelligence Tools (IJAIT)* 17.4, pp. 745–763.
- Greiner, R., B. A. Smith, and R. W. Wilkerson (1989). “A correction to the algorithm in Reiter’s theory of diagnosis”. In: *Artificial Intelligence* 41.1, pp. 79–88.
- Grimm, S. et al. (2011). “Elimination of Redundancy in Ontologies”. In: *The Semantic Web: Research and Applications*. Springer, pp. 260–274.
- Gupta, S.D., B. Genc, and B. O’Sullivan (2021). “Explanation in Constraint Satisfaction: A Survey”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21)*. Survey Track. International Joint Conferences on Artificial Intelligence Organization, pp. 4400–4407.
- Hentze, M. et al. (2021). “Hyper Explanations for Feature-Model Defect Analysis”. In: *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. Krems, Austria: Association for Computing Machinery.
- Heradio, R. et al. (2022). “Uniform and scalable sampling of highly configurable systems”. In: *Empirical Software Engineering* 27.2, p. 44.
- Hotz, L. et al. (2014). “Configuration Knowledge Representation and Reasoning”. In: *Knowledge-Based Configuration: From Research to Business Cases*. Ed. by A. Felfernig et al. Boston: Morgan Kaufmann, pp. 41–72.
- Ignatiev, A., A. Morgado, and J. Marques-Silva (2018). “PySAT: A Python Toolkit for Prototyping with SAT Oracles”. In: *Theory and Applications of Satisfiability Testing – SAT 2018*, pp. 428–437.
- Jannach, D., T. Schmitz, and K. Shchekotykhin (2015). “Parallelized Hitting Set Computation for Model-Based Diagnosis”. In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence*. Austin, Texas: AAAI Press, pp. 1503–1510.
- Jannach, D., T. Schmitz, and K. Shchekotykhin (2016). “Parallel Model-Based Diagnosis on Multi-Core Computers”. In: *Journal of Artificial Intelligence Research* 55, pp. 835–887.
- Junker, U. (2004). “QUICKXPLAIN: Preferred Explanations and Relaxations for over-Constrained Problems”. In: *Proceedings of the 19th National Confer-*

- ence on Artificial Intelligence. AAAI'04. San Jose, California: AAAI Press, pp. 167–172.
- Junker, U. (2006). “Configuration”. In: *Handbook of Constraint Programming*. Ed. by F. Rossi, P. van Beek, and T. Walsh. Amsterdam, The Netherlands: Elsevier, pp. 837–873.
- Kang, K. et al. (1990). *Feature-oriented Domain Analysis (FODA) – Feasibility Study*. Tech. rep. SEI-90-TR-21. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University.
- Katoch, S., S.S. Chauhan, and V. Kumar (2021). “A review on genetic algorithm: past, present, and future”. In: *Multimedia Tools and Applications* 80.5, pp. 8091–8126.
- Kim, C. et al. (2011). “Reducing Combinatorics in Testing Product Lines”. In: AOSD '11. Porto de Galinhas, Brazil: Association for Computing Machinery, pp. 57–68.
- de Kleer, J. and B.C. Williams (1987). “Diagnosing multiple faults”. In: *Artificial Intelligence* 32.1, pp. 97–130.
- Kocsis, Levente and Csaba Szepesvári (2006). “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML 2006*. Vol. 4212. Lecture Notes in Computer Science. Springer, pp. 282–293.
- Kowal, M., S. Ananieva, and T. Thüm (2016). “Explaining Anomalies in Feature Models”. In: *ACM SIGPLAN Notices* 52.3, pp. 132–143.
- Le, D., H. Lee, et al. (2013). “Validating Consistency between a Feature Model and Its Implementation”. In: *Safe and Secure Software Reuse*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–16.
- Le, V.M., T.N.T. Tran, and A. Felfernig (2021). “A Conversion of Feature Models into an Executable Representation in Microsoft Excel”. In: *Intelligent Systems in Industrial Applications*. Springer International Publishing, pp. 153–168.
- Le, V.M., A. Felfernig, M. Uta, D. Benavides, et al. (2021). “DIRECTDEBUG: Automated Testing and Debugging of Feature Models”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 81–85.
- Le, V.M., A. Felfernig, T.N.T. Tran, Müslüm Atas, et al. (2021). “DIRECTDEBUG: A software package for the automated testing and debugging of feature models”. In: *Software Impacts* 9, p. 100085.
- Le, V.M., A. Felfernig, and T.N.T. Tran (2022). “Test case aggregation for efficient feature model testing”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B*. SPLC '22. Graz, Austria: Association for Computing Machinery, pp. 174–177.
- Le, V.M., A. Felfernig, M. Uta, T.N.T. Tran, et al. (2022). “WIPEOUTR: automated redundancy detection for feature models”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*. SPLC '22. Graz, Austria: Association for Computing Machinery, pp. 164–169.
- Le, V.M., C. Vidal Silva, et al. (2023). “FASTDIAGP: An Algorithm for Parallelized Direct Diagnosis”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37.5, pp. 6442–6449.
- Le, V.M., A. Felfernig, T.N.T. Tran, and M. Uta (2024). “INFORMEDQX: Informed Conflict Detection for Over-Constrained Problems”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.9, pp. 10616–10623.

- Le, V.M., T.N.T. Tran, S. Lubos, et al. (2026). “Early-Stage Product Line Validation Using LLMs: A Study on Semi-Formal Blueprint Analysis”. In: *Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing*. ACM, pp. 1620–1627.
- Le, V.M., L.A. Feldgrill, and A. Felfernig (2026). “Robust Lazy Conflict Detection via Multi-Conflict Extraction and Genetic Diversity Control”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 40.23, pp. 19208–19215.
- Le Berre, D. and A. Parrain (2010). “The SAT4J library, release 2.2”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7.2-3, pp. 59–64.
- Liffiton, M. and K. Sakallah (2008). “Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints”. In: *Journal of Automated Reasoning* 40, pp. 1–33.
- Marques-Silva, J., I. Lynce, and S. Malik (2009). “Conflict-Driven Clause Learning SAT Solvers”. In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 131–153.
- Marques-Silva, J., F. Heras, et al. (2013). “On Computing Minimal Correction Subsets”. In: *23rd International Joint Conference on Artificial Intelligence*. Beijing, China, pp. 615–622.
- Marques-Silva, J. and A. Previti (2014). “On computing preferred MUSes and MCSes”. In: *17th International Conference on Theory and Applications of Satisfiability Testing (SAT-2014)*, pp. 58–74.
- McGuinness, D. (2007). “Configuration”. In: *The Description Logic Handbook: Theory, Implementation and Applications*. 2nd ed. Cambridge University Press, pp. 417–435.
- Mendonca, M., A. Wąsowski, and K. Czarnecki (2009). “SAT-Based Analysis of Feature Models is Easy”. In: *Proceedings of the 13th International Software Product Line Conference*. San Francisco, CA, USA, pp. 231–240.
- Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press.
- Mittal, S. and F. Frayman (1989). “Towards a Generic Model of Configuration Tasks”. In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2. IJCAI’89*. Detroit, Michigan: Morgan Kaufmann Publishers Inc., pp. 1395–1401.
- Mostafa, S.A. and I.A. Ahmad (2018). “Recent developments in systematic sampling: A review”. In: *Journal of Statistical Theory and Practice* 12.2, pp. 290–310.
- Nica, I. et al. (2013). “The Route to Success – A Performance Comparison of Diagnosis Algorithms”. In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pp. 1039–1045.
- Nieke, M. et al. (2018). “Anomaly Analyses for Feature-Model Evolution”. In: *ACM SIGPLAN Notices* 53.9, pp. 188–201.
- Nielsen, J. (1994). *Usability engineering*. Morgan Kaufmann.
- O’Neil, P.E. (1987). “Model 204 architecture and faperformance”. In: *International Workshop on High Performance Transaction Systems*. Springer, pp. 39–59.
- O’Sullivan, B. et al. (2007). “Representative Explanations for Over-Constrained Problems”. In: *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*. Vancouver, Canada, pp. 323–328.

- Paulraj, S. and P. Sumathi (2010). “A Comparative Study of Redundant Constraints Identification Methods in Linear Programming Problems”. In: *Mathematical Problems in Engineering*, pp. 1–16.
- Piette, C. (2008). “Let the Solver Deal with Redundancy”. In: *20th IEEE International Conference on Tools with Artificial Intelligence*, pp. 67–73.
- Popescu, A. et al. (2022). “An Overview of Machine Learning Techniques in Constraint Solving”. In: *Journal of Intelligent Information Systems* 58.1, pp. 91–118.
- Prosser, P. (1993). “Hybrid Algorithms for the Constraint Satisfaction Problem”. In: *Computational Intelligence* 9.3, pp. 268–299.
- Prud’homme, C. and J.-G. Fages (2022). “Choco-solver: A Java library for constraint programming”. In: *Journal of Open Source Software* 7.78, p. 4708.
- Reiter, R. (1987). “A theory of diagnosis from first principles”. In: *Artificial Intelligence* 32.1, pp. 57–95.
- Rodas-Silva, J. et al. (2019). “Selection of software product line implementation components using recommender systems: An application to wordpress”. In: *IEEE Access* 7, pp. 69226–69245.
- Rodler, P. (2022). “A Formal Proof and Simple Explanation of the QuickXplain Algorithm”. In: *Artificial Intelligence Review* 55.8, pp. 6185–6206.
- Rossi, F., P. van Beek, and T. Walsh (2006). *Handbook of Constraint Programming*. Elsevier Science.
- Rossi, F., K. Venable, and T. Walsh (2011). *A Short Introduction to Preferences: Between Artificial Intelligence and Social Choice*. Morgan & Claypool Publishers.
- Russell, S.J. and P. Norvig (2021). *Artificial Intelligence: A Modern Approach*. 4th. Pearson.
- Sabin, M. and E. Freuder (1999). “Detecting and Resolving Inconsistency and Redundancy in Conditional Constraint Satisfaction Problems”. In: *AAAI Workshop on Configuration*. Orlando, FL, USA, pp. 90–94.
- Schobbens, P.-Y., P. Heymans, and J.-C. Trigaux (2006). “Feature Diagrams: A Survey and a Formal Semantics”. In: *14th IEEE International Conference on Requirements Engineering*, pp. 139–148.
- Segura, S. et al. (2012). “BETTY: Benchmarking and Testing on the Automated Analysis of Feature Models”. In: *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems*. Leipzig, Germany: ACM, pp. 63–71.
- Shchekotykhin, K., G. Friedrich, et al. (2012). “Interactive Ontology Debugging: Two Query Strategies for Efficient Fault Localization”. In: *Journal of Web Semantics* 12–13, pp. 88–103.
- Shchekotykhin, K., D. Jannach, and T. Schmitz (2015). “MergeXPlain: Fast Computation of Multiple Conflicts for Diagnosis”. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI’15. Buenos Aires, Argentina: AAAI Press, pp. 3221–3228.
- She, S. et al. (2010). “The Variability Model of The Linux Kernel.” In: *VaMoS* 10.10, pp. 45–51.
- Sinz, C., A. Kaiser, and W. Küchlin (2003). “Formal methods for the validation of automotive product configuration data”. In: *Ai Edam* 17.1, pp. 75–97.
- Stumptner, M. (1997). “An Overview of Knowledge-based Configuration”. In: *AI Communications* 10.2, pp. 111–125.

- Tran, Son N. and Artur d'Avila Garcez (2023). "Neurosymbolic Reasoning and Learning with Restricted Boltzmann Machines". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 5, pp. 6558–6565.
- Trinidad, P. et al. (2008). "Automated error analysis for the agilization of feature modeling". In: *Journal of Systems and Software* 81 (6), pp. 883–896.
- Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Acad. Press.
- Uran, C. and A. Felfernig (2018). "Lazy Conflict Detection with Genetic Algorithms". In: *Recent Trends and Future Technology in Applied Intelligence*. Cham: Springer International Publishing, pp. 175–186.
- Harmelen, F. van, V. Lifschitz, and B. Porter (2008). *Handbook of Knowledge Representation*. Foundations of Artificial Intelligence. Elsevier.
- Vidal, C. et al. (2021). "Explanations for over-constrained problems using QuickXPlain with speculative executions". In: *Journal of Intelligent Information Systems* 57.3, pp. 491–508.
- Walsh, T. (2007). "Representing and Reasoning with Preferences". In: *AI Magazine* 28.4, pp. 59–70.
- White, J. et al. (2010). "Automated diagnosis of feature model configurations". In: *Journal of Systems and Software* 83.7, pp. 1094–1107.
- Zeller, A. (2001). "Automated debugging: are we close?" In: *Computer* 34.11, pp. 26–31.
- Zeller, A. and R. Hildebrandt (2002). "Simplifying and Isolating Failure-Inducing Input". In: *IEEE Transactions on Software Engineering* 28.2, pp. 183–200.