A Comparison of Strategies for Tolerating Inconsistencies during Decision-Making

Alexander Nöhrer
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
alexander.noehrer@jku.at

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University
Linz, Austria
armin.biere@jku.at

Alexander Egyed
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT

Tolerating inconsistencies is well accepted in design modeling because it is often neither obvious how to fix an inconsistency nor important to do so right away. However, there are technical reasons why inconsistencies are not tolerated in many areas of software engineering. The most obvious being that common reasoning engines are rendered (partially) useless in the presence of inconsistencies. This paper investigates automated strategies for *tolerating inconsistencies* during decision-making in product line engineering, based on isolating parts from reasoning that cause inconsistencies. We compare trade offs concerning incorrect and incomplete reasoning and demonstrate that it is even possible to fully eliminate incorrect reasoning in the presence of inconsistencies at the expense of marginally less complete reasoning. Our evaluation is based on seven medium-to-large size software product line case studies. It is important to note that our mechanism for tolerating inconsistencies can be applied to arbitrary SAT problems and thus the basic principles of this approach are applicable to other domains also.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Algorithms, Human Factors, Verification

Keywords

Formal Reasoning, User Guidance, Inconsistencies

1. INTRODUCTION

Inconsistencies in models imply the presence of errors. For the software engineer, the main benefit of tolerating inconsistencies is the ability to continue working despite this presence of errors. This is useful when it is neither obvious how to fix the inconsistency nor important to do so right away. Indeed, many inconsistencies can be tolerated. Balzer argued that inconsistencies should be detected and communicated to the developers; however, developers should not be hindered in continuing their work despite the presence of inconsistencies [1]. The same is true in software product line engineering. For instance during the configuration of software components there could be an inconsistency concerning the communication protocol between components. However, the protocol could have a dependency to other component decisions which the engineer might prefer to explore before deciding on the protocol. Another example would be if an engineer starts with an existing configuration and adapts it. Changing decisions might cause inconsistencies that could be fixed by follow-on decisions, however the engineer should not be hindered to make other decisions in the meantime. Also during the creation of feature models it would be desirable to have automations working that inform the engineer e. g. about dead features despite the feature model not being consistent at the moment.

While in model-driven software engineering, it is state-of-the-practice to tolerate inconsistencies [6], most product configuration tools, disallow tolerating inconsistencies (i. e., usually by preventing decisions that cause inconsistencies). And there are good reasons for disallowing inconsistencies. First and foremost, many reasoning engines are rendered useless in the presence of inconsistencies. Even if the reasoning engines were to function (instead of failing outright), the implications on the quality of the results are typically not understood (conventional wisdom implies that we cannot expect a reasoning engine to compute correct results in the presence of inconsistent, aka erroneous input). This is a severe problem because as Balzer said, inconsistencies are a fact of (software engineering) life and to date reasoning engines enable many vital automations during product configuration. When we speak of automations during product configuration, we speak of any automations that are conceivable based on reasoning with the model and the configuration state, such as: *i)* understanding the effects of decisions, *ii)* deducing other decisions, *iii)* determining the optimal order of questions [19] etc. Typically such automations fail, if the reasoning engine fails; or they produce incorrect results if the reasoning is incorrect.

As a continuation of our previous work [17], the goal of this paper is to investigate how to handle inconsistencies during product configuration based on SAT-based reasoning [4] while still benefiting from a definable level of completeness and correctness. Contrary to conventional wisdom, we will demonstrate that *i) correct reasoning in the presence of in-*

SPLC'12, September 02–07, 2012, Salvador, Brazil.

consistencies is possible and *ii)* automations (SAT-based) remain useful even while tolerating inconsistencies. In contrast to our previous work [17], were we investigated the applications of one technique to manage inconsistencies in different areas of product line engineering, we extended the number of techniques but narrowed the focus to one application to do a in depth evaluation. Even though the evaluation focuses on product configuration, one decision at a time, this work should be applicable to other configuration processes, such as starting with a default product.

While this paper focuses on product configuration, it is the first study to compare strategies for tolerating inconsistencies in SAT-based reasoning. As such, we believe that this work has wider applicability and may benefit other areas in product line engineering and in software engineering in general, that rely on SAT-based reasoning, e. g. feature model analysis, SAT-based hardware verification [9] or debugging [21].

Next, we present an illustration used throughout the paper. We discuss the goals of this work and the challenges. A detailed discussion of the tolerating strategies follows, including a discussion of trade-offs. The evaluation, related work, future work and conclusions round off the paper. It is out of the scope of this paper to discuss how to fix inconsistencies with or without tolerating them; however we believe that tolerating inconsistencies also makes the fixing of such inconsistencies easier – an observation that is alluded at various places in the paper, but future work to explore and already partially covered in our paper [17].

2. PROBLEM DESCRIPTION

Since this paper investigates how to tolerate inconsistencies while reasoning with *SAT solvers*, a short introduction to the terminology based on [3] is given: SAT problems are defined in conjunctive normal form (CNF) which is a conjunction of clauses. One *clause* is a disjunction of *literals* which are Boolean variables. *Assumptions* are assignments for literals that constrain the assignment possibility of a literal to either true or false, such assumptions can either be derived or made by the user. SAT solvers produce one of two results, either a CNF is satisfiable (SAT) or unsatisfiable ($UNSAT$) – SAT meaning that there exists an assignment for all literals such that the CNF evaluates to true, UNSAT meaning that such an assignment does not exist. If a problem is UNSAT it can be because of either an inconsistency in the clauses which would be *low-level*, or an inconsistency because of assumptions which would be *high-level*.

2.1 Illustrative Configuration Example

As an illustrative configuration example we will be using an excerpt from a real e-commerce decision-oriented product line. This decision model was reverse engineered from the the DELL homepage (during February 2009), a complete version of the model was also used in the evaluation and it can be downloaded from the C2O website[1]. The illustration's relevant questions (e. g. $Screen\,Size$) with their respective choices (e. g. $12.1''$, $13.3''$, and $15.4''$) are shown in Figure 1, the indicated constraint relations are listed next:

$$(Screen\,Size \Rightarrow Laptop\,Type)_{ConstraintRelation} := \{$$
$$(12.1'' \Rightarrow \{Inspirion, Latitude\}),$$
$$(13.3'' \Rightarrow \{Latitude, Vostro\}),$$
$$(15.4'' \Rightarrow \{Inspirion, Latitude, Vostro\})\}$$

[1] www.sea.jku.at/tools/c2o

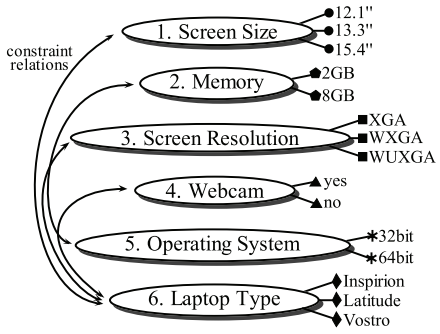

Figure 1: Illustrative Excerpt of a Product Line Decision Model.

$$(Memory \Rightarrow Operating\,System)_{ConstraintRelation} := \{$$
$$(2GB \Rightarrow \{32bit, 64bit\}),$$
$$(8GB \Rightarrow \{64bit\})\}$$

$$(Screen\,Resolution \Rightarrow Laptop\,Type)_{ConstraintRelation} := \{$$
$$(XGA \Rightarrow \{Latitude, Vostro\}),$$
$$(WXGA \Rightarrow \{Inspirion, Latitude, Vostro\}),$$
$$(WUXGA \Rightarrow \{Latitude, Vostro\})\}$$

$$(Webcam \Rightarrow Laptop\,Type)_{ConstraintRelation} := \{$$
$$(yes \Rightarrow \{Inspirion, Vostro\}),$$
$$(no \Rightarrow \{Latitude, Vostro\})\}$$

These relations impose constraints onto a configuration, such as selecting $2GB$ of $Memory$, both $32bit$ and $64bit$ $Operating\,Systems$ are viable, but if $8GB$ is selected only a $64bit$ $Operating\,System$ is allowed.

Encoding this example in CNF is straight-forward, by assigning a literal to each choice (e. g. $Memory(2GB) \rightarrow a$, $Memory(8GB) \rightarrow b$, $Operating\,System(32bit) \rightarrow c$), preserving the question concepts through clauses that only allow one choice per question (e. g. for the Memory question: $(\neg a \vee \neg b) \wedge (a \vee b)$), and adding clauses for the constraint relations (e. g. the relation between Memory and Operating System: $(\neg b \vee \neg c)$).

2.2 Terminology

In our case of decision-making scenarios supported with SAT-based reasoning, one has to distinguish between the user's perspective and that from the reasoning engine. Users make decisions in a configuration tool (e. g. our C2O configuration tool [18]), these decisions are then translated to user assumptions to be used in the SAT-based reasoning engine which then further derives assumptions.

2.3 Reasoning during Product Configuration

In the domain of product configuration and decision modeling, SAT-based reasoning is state-of-the-practice [15, 22]. It has several primary uses: SAT reasoning is used *i)* to validate products, *ii)* to find viable alternative solutions if a product is not valid [24] or auto complete partial products [15], and *iii)* to provide guidance during the configuration process [20, 19]. To validate a product one call to the SAT solver is sufficient. For the other uses several SAT solver calls are necessary with different assumptions to find out if combinations of assumptions are valid. So basically in these cases the SAT solver is used as an oracle and the reasoning process is based on querying this oracle.

The most basic guidance that SAT solvers are used for in decision-making scenarios, is to calculate the effect of a decision and show its effects to the user when subsequent

Table 1: Configuration Progression of the Example given in Figure 1 at the Literal Level.

Choice	$1^{st}q$		$2^{nd}q$		$3^{rd}q$		$4^{th}q$
	u	s	u	s	u	s	u
Screen Size(12.1")	**1**	**1**		**1**		**1**	
Screen Size(13.3")		0		0		0	
Screen Size(15.4")		0		0		0	
Memory(2GB)				0		0	
Memory(8GB)			**1**	**1**		**1**	
Screen Res.(XGA)					**1**	**1**	
Screen Res.(WXGA)						0	
Screen Res.(WUXGA)						0	
Webcam(yes)						0	**1?**
Webcam(no)						1	
OS(32bit)				0		0	
OS(64bit)				1		1	
Laptop Type(Inspirion)						0	
Laptop Type(Latitude)						1	
Laptop Type(Vostro)		0		0		0	

q...question, u...user assumption, s...derived state, 0...false, 1...true

questions are answered. This is illustrated in Table 1, for the first question the user decided the $Screen\,Size$ to be $12.1''$ which is indicated in the column $1^{st}q$, u, the effect and resulting state of this decision is shown in the column $1^{st}q$, s. This is repeated for the other questions, where user assumptions are bold and choices belonging to questions that already have been answered are grayed out. The effect of each user assumption is calculated the following way: For every remaining literal a SAT call with a positive as well as a negative assumption is made, to see if there are any solutions left with those assumptions. If both assumptions are still possible the answer had no effect on this literal at that time, but if only one assumption is possible this assumption can be kept as a derived assumption (e. g. the assumptions $Screen\,Size(12.1'')$ and $Screen\,Size(13.3'')$ are UNSAT, therefore $\neg Screen\,Size(13.3'')$ is derived, the same is true for $\neg LaptopType(Vostro)$). This process is repeated after each decision during the whole process.

In the illustration in Table 1 technically after the third decision the decision-making process could be stopped, since an assumption was provided or derived for all literals. However what if the user is not satisfied with some of the derived decisions as illustrated and wants to select $Webcam(yes)$? There are three possibilities, shown in Figure 2, how to handle such a situation: *i)* not allowing such a decision and thereby forcing the user to backtrack and explore different decisions so that $Webcam(yes)$ becomes available, *ii)* fixing the problem right away by deciding differently for an earlier question that $Webcam(yes)$ is in conflict with, and *iii)* tolerate the inconsistency for the time being until the user made up his / her mind on how to fix it.

In this paper, we focus on tolerating inconsistencies as a viable alternative to disallowing inconsistencies or forcing an immediate fix. Our tolerating mechanisms are based on isolating contributing assumptions of an inconsistency from reasoning, however this is hidden from the user, the decisions are not discarded and the configuration process can be continued in case of an inconsistency with no limitations. We believe an indication of isolated decisions and not keeping the user from making new decisions is preferable, so the user can decide when to fix an inconsistency. One also has to separate the user's view from the reasoning behind, e. g. users are still able to see what contradictory decisions they made even though they might be excluded from reasoning. For the remainder of this paper we solely focus on tolerating mechanisms from the perspective of a SAT-based reasoning engine.

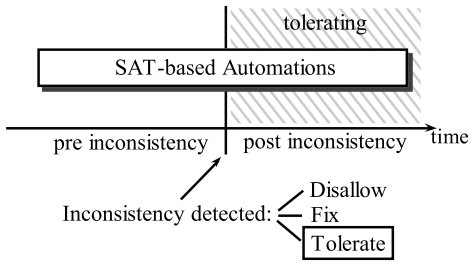

Figure 2: Overview of SAT-based reasoning.

2.4 Reasoning in the Presence of Inconsistencies

As long as a CNF with assumptions evaluates to SAT, no inconsistency is detected. Adding additional clauses and / or assumptions may change SAT to UNSAT, but once the SAT solver is in an UNSAT (inconsistent) state adding additional clauses and / or assumptions will have no effect at all because the SAT solver will continue to evaluate to UNSAT. As explained in Section 2.3, since the SAT solver is used for more then just detecting inconsistencies, automations are lost too. For example, the ability to automatically derive assumptions *i)* to (partially) auto-complete the configuration process or *ii)* show decision effects, is lost.

If the tolerance to inconsistencies should not change the SAT-based automation then the only option is isolation. It is important to distinguish between isolating and fixing an inconsistency at this point: Isolating means sandboxing clauses and / or assumptions that cause an inconsistency, in other words identify contributors and ignore them for reasoning purposes. Fixing would go one step further and in addition change those clauses and / or assumptions in such a way the inconsistency would be resolved. So the isolation can be seen as a first step of an actual fix without committing on how to fix.

In the domain of decision-making, isolating clauses and assumptions account for different parts. Assumptions are used to express user decisions and derived decisions (high-level), whereas clauses are used to define the decision model (low-level). This paper works under the assumption that the decision model is correct, and that the user decisions are contradictory to the model (much like we presume that in case of inconsistencies in design models, the user model is at fault and not the meta model that defines the modeling language). Therefore we only care about isolating user assumptions, ignoring the clauses. So looking at the example from Table 1 with the inconsistency detected at the fourth answer, given the constraints and considering all possibilities, the following decisions contribute to the inconsistency: $Screen\,Size(12.1'')$, $Screen\,Resolution(XGA)$ and $Webcam(yes)$. As a matter of fact isolating any one of these is sufficient to get meaningful results again. At this point the user could continue configuring the product and fix the problem later on. Once the user decides to fix the problem,

the defective contributor has to be identified by the user, since choosing a random contributor often does not suffice. The user then has to provide a different, valid decision for the identified question, resulting in the fix of the problem. The fixing aspect is out of the scope of this work.

3. GOALS

Our goal is to provide SAT-based automations (i. e., guidance in our case), that remain as complete and as correct as possible in the presence of inconsistencies. We discussed the advantages of tolerating inconsistencies in the introduction and we discussed that tolerating inconsistencies should not affect the SAT-based automations themselves (ideally SAT-based automations should remain the same regardless of whether inconsistencies are tolerated or not). Tolerating inconsistencies without changing SAT-based automations is doable by isolating offending assumptions. Since there are alternatives on how to isolate them we will investigate four isolation strategies and compare their advantages and disadvantages in terms of incorrect reasoning and incomplete reasoning, which will be explained in detail later. Some of the isolation strategies are intuitive (even trivial) others may not be known to the software engineering community. A secondary goal of this paper is thus to raise awareness of strategies less known – particularly HUMUS. We make no claim that the four isolation strategies represent a complete set but believe that they cover interesting ends of the problem and solution spectrum.

4. APPROACH

In this section we will discuss different isolation strategies in detail. However, as mentioned before in Section 2.4, we will solely focus on isolating user assumptions and recalculating derived assumptions (high-level facts during SAT-based reasoning) and assume that the decision models themselves are correct and therefore the clauses (low-level facts during SAT-based reasoning) do not need to be changed. However, it should be pointed out that two out of the four isolation strategies (MaxSAT and HUMUS) could also be applied to isolate low-level facts [13, 12].

To provide a better insight into the investigated SAT isolation techniques we need to introduce a few more (high-level) SAT concepts [3, 13]. First an important SAT concept is a minimal unsatisfiable set (MUS) which is defined by the properties of being minimal and that removing any single assumption results in the remaining set being satisfiable. In our example configuration illustrated in Table 1 only one MUS of user assumptions is present $\{Screen\,Size(12.1'')$, $Screen\,Resolution(XGA)$, $Webcam(yes)\}$. However generally speaking inconsistencies consist of many possibly overlapping MUSes, as a consequence isolating one assumption of one MUS does not necessarily result in a satisfiable SAT model. Another concept is the minimal correcting set (MCS) which is defined by the properties of being minimal and that removing it from reasoning, results in a satisfiable SAT model. In our illustration the MCSes are $\{Webcam(yes)\}$, $\{Screen\,Resolution(XGA)\}$, and $\{Screen\,Size(12.1'')\}$. Generally speaking MUSes and MCSes are connected via hitting sets, meaning that every MCS is composed of a single element from every MUS. In addition to this relation MCSes are the complement of a maximal satisfiable set (MSS). As the name already states a MSS is a set of assumptions that is satisfiable and of the maximum size it can be. An example for one possible MCS and its complementary MSS given our illustration, is the MCS $\{Screen\,Size(12.1'')\}$ and the MSS $\{Memory(8GB), Screen\,Resolution(XGA), Webcam(yes)\}$.

Table 2: Isolation strategies based on the Example in Table 1.

Choice	$3^{rd}q$	$4^{th}q$				
			Disregard all	*Skip*	*MaxSAT*	*HUMUS*
	s	u	s	s	s	s
Screen Size(12.1")	**1**			**1**	**1**	
Screen Size(13.3")	0			0	0	
Screen Size(15.4")	0			0	0	
Memory(2GB)	0			0	0	0
Memory(8GB)	**1**			**1**	**1**	**1**
Screen Res.(XGA)	**1**			**1**	0	
Screen Res.(WXGA)	0			0	1	
Screen Res.(WUXGA)	0			0	0	
Webcam(yes)	0	**1?**		0	**1**	
Webcam(no)	1			1	0	
OS(32bit)	0			0	0	0
OS(64bit)	1			1	1	1
Laptop Type(Inspirion)	0			0	1	
Laptop Type(Latitude)	1			1	0	
Laptop Type(Vostro)	0			0	0	

q...question, u...user assumption, s...derived state, 0...false, 1...true

4.1 Different Isolation Strategies

In the following the four different isolation strategies we investigated will be explained in detail, with the help of Table 2. This table continues the configuration example from Table 1 where the derived state after the third questions was complete, but introduces a conflicting user assumption at the fourth question. At this point depending on the isolation strategy different alternative states can be derived, which are explained next.

4.1.1 Disregard All Strategy

A trivial way to ensure a correct state in the presence of inconsistencies is to ignore everything that happened so far, meaning that every single assumption before the inconsistency was encountered is isolated from future reasoning. This solution may seem mundane; however, we can think of it as the worst-case strategy against which others can be compared. This strategy is also specific to high-level isolation since isolating all clauses hardly makes sense. The result of this isolation strategy can be seen in Table 2: All user assumptions are isolated resulting in no derived assumptions for future questions, but also for already decided questions – basically the initial state is restored.

4.1.2 Skip Strategy

In order to be consistent again, this strategy skips the user assumption immediately preceding the detection of the inconsistency. Unless clauses are added iteratively this also makes no sense for a low-level isolation strategy (hence it applies to high-level facts only). The result of this isolation strategy can be seen in Table 2: The conflicting user as-

sumption $Webcam(yes)$ is isolated (skipped) – basically the state before the inconsistency detection is kept, requiring a simple history of user assumptions to implement it.

4.1.3 MaxSAT Strategy

MaxSAT stands for maximum satisfiability of the highest cardinality [12], the basic concept is to identify a set of clauses that can be satisfied and consists of the maximum number of clauses possible. In our case since we do not care about clauses but assumptions, this idea can be translated to keeping as many user assumptions as possible that do not contradict each other, or in other words find the "closest" solution. After a MaxSAT solution is calculated, every assumption not contained in the solution is isolated. While Disregard All and Skip are isolation strategies with a single solution, MaxSAT is different in that there could be multiple, alternative "closest" solutions with an equal number of user assumptions kept. For the given illustration there are three possible solutions of the same cardinality, since MaxSAT is non-deterministic any of those alternatives can be the result. The one solution presented in Table 2 isolates the user assumption $Screen\,Resolution(XGA)$, the most noticeable effects of this isolation are that the derived positive assumption $Screen\,Resolution(WXGA)$ and that $LaptopType(Inspirion)$ is derived instead of $LaptopType(Latitude)$. The isolation solution from Skip is a special case of a MaxSAT isolation; however, it may not always be desired to isolate the last decision made by the user.

The implementation of the MaxSAT strategy is realized by searching for a MSS with maximum cardinality, starting at a cardinality of the number of user assumptions minus one and decreasing it by one if no such set is found. Once a MSS with a maximum cardinality is found every assumption that is not contained in this set is isolated. MaxSAT typically returns the first MSS it finds, ignoring potential other MSS of the same size.

4.1.4 HUMUS Strategy

HUMUS stands for High-level Union of Minimal Unsatisfiable Sets, it is a concept based on the calculation of all Minimal Unsatisfiable Sets (MUSes) [13], which again targets more at the low-level. As already mentioned in our paper [17], the basic concept behind it is to isolate all contributors (directly and indirectly) of the inconsistency and only keep assumptions that have no relation to the inconsistency. The result of this isolation strategy is depicted in Table 2, by isolating all contributors the only impartial assumption $Memory(8GB)$ is kept. Note that the HUMUS calculation only returns a single result like Skip or Disregard All because a user assumption either contributes to the inconsistency (in which case it is in the HUMUS) or it does not.

The implementation of the HUMUS strategy takes a shortcut in comparison to the approach of Liffiton to compute all MUSes [13], since we only care about the union of the MUSes and not the individual MUSes themselves. Our implementation uses a variant of Liffiton's [13] approach to calculate MSSes using assumptions over clause selector variables. If a satisfiable subset of clauses has been found, which cannot be increased in size without making the resulting formula unsatisfiable, the complement of this set is an MCS. This particular satisfiable subset is then blocked with a blocking clause. We do not use at-most constraints, this avoids having to reset the SAT solver as soon as an MCS of a different size is found. After having calculated all the MCSes we simply calculate the union of the MCSes, since the resulting set is the same as the union of all MUSes due to the relation via hitting sets. The exact implementation details are omitted here due to space limitations.

4.2 Discussion of Isolation Strategies

As can be observed in Table 2 the different isolation strategies, described in the last section, result in different states after the inconsistency is encountered. Thus they do not achieve the same result but rather provide alternatives on how to proceed after an inconsistency is encountered. All strategies have in common that the SAT-based automations appear functional again (SAT is returned instead of UNSAT). But how can one tell which resulting state is the most complete or correct one, without knowing how the user eventually will fix the inconsistency? Obviously there must be qualitative differences once the user fix is known and the reasoning so far can be analyzed with respect to the now known fix. These qualitative differences can be divided into three categories and are discussed next.

4.2.1 Incomplete Reasoning

If the isolation strategy removes correct user assumptions then the reasoning gets incomplete due to missing information in the reasoning process (not as much is inferred as could be). As a result user guidance (see Section 2.3) would potentially offer fewer derived assumptions. For decision-making scenarios, fewer derived assumptions are not problematic except that the degree of automation decreases (hence, more isolation implies less automation). However, do note that that completeness during user guidance is a loose concept. For example at the beginning of the configuration process, no user guidance is available, because no user assumptions are available to reason with. During the configuration process, depending on how many relations the given answers are in, the number of derived assumptions increases steadily. And at the end of the configuration process when all is known, naturally guidance would be best since the reasoning is complete, but that is also the point where guidance is not needed anymore.

Generally speaking as can already be observed in Section 4.1, the Disregard All isolation strategy results in the most incomplete reasoning possible (worst case), while the MaxSAT and Skip strategies potentially suffer the least incomplete reasoning (best case). For the HUMUS strategy the degree of incomplete reasoning could vary between the best and the worst case depending on the number of constraints in the model and therefore the number of involved assumptions. However many constraints in a model could indicate overlapping constraints and redundant information. For example revisiting the illustration in Figure 1, both $Screen\,Size(13.3'')$ and $Screen\,Resolution(WUXGA)$ result in the elimination of $LaptopType(Inspirion)$. So isolating only one of those two assumptions would not result in incomplete reasoning (hence, isolation results in a potential incompleteness only). Even if both assumptions would be isolated and cause incompleteness due to something correct being isolated, new answers like $Webcam(no)$ would reprovide this lost piece of information again (hence, isolation may lead to temporary incompleteness only).

4.2.2 Incorrect Reasoning

Incorrect reasoning is the result of reasoning with defects. Although we do not investigate fixes in this paper, they are crucial for determining incorrect reasoning because

they identify defects that caused inconsistencies. Incorrect reasoning can be determined by analyzing the effects that defects have on the reasoning process, if not isolated. Reasoning with defects would mean that the guidance might leave out correct choices or even suggest incorrect choices to follow-up questions. Since constraints are somewhat redundant as was discussed above, a non-isolated defect might also lead to another inconsistency later. That is, it may conflict with new user assumptions while tolerating inconsistencies which seems more equivalent to postponing inconsistencies rather than tolerating inconsistencies. Related to this problem is the detection of another inconsistency while already tolerating an inconsistency, in this case it cannot be determined if it is an inconsistency related to the inconsistency that was tolerated earlier or if it is in fact a new, unrelated inconsistency.

When using the Disregard All and HUMUS isolation strategies one can be sure to eliminate the defect, because Disregard All isolates all assumptions made by the user prior to the inconsistency (and is as such conservative) and HUMUS computes all assumptions involved in the inconsistency (directly and indirectly). HUMUS, in the worst case, could isolate everything like Disregard All if all assumptions are contributors to the inconsistency. On the other hand with the MaxSAT and Skip strategies it is the inconsistency that is eliminated and not necessarily the assumption(s) that the user will change later when fixing (though by random chance these strategies may also isolate these assumptions). It follows that one cannot be sure if the configuration process is being continued with incorrect derived assumptions based on something that will be fixed and hence the user should not fully trust the results derived from these kinds of automations. In other words MaxSAT and Skip are maximizing what assumptions to keep (they isolate less) which likely leads to less incomplete reasoning, though at the expense of incorrect reasoning. Disregard All and HUMUS likely lead to more incomplete reasoning, though they are guaranteed not to lead to incorrect reasoning – in the presence of inconsistencies.

4.2.3 Revisitation

Depending on the use of the SAT solver this could be more or less important. In decision-making scenarios this means that the more user assumptions are isolated the more questions have to be potentially revisited at a later point in time (less automation for the end user). As with incomplete reasoning this issue is the biggest for the Disregard All approach, the smallest for the MaxSAT and Skip approaches, and highly depends on the model and situation for the HUMUS approach.

5. EVALUATION

To assess the differences of isolation strategies, we evaluated them on seven product line and decision models from various domains (e-commerce, decision models, feature models). The models were different in size and complexity. For example, the *Dell1* e-commerce model (only a very simplified version thereof was used as illustration in this paper) had 28 questions, with roughly 5 choices per question, and 111 relations. We also investigated another version of the *Dell1* model, the *Dell2* model, using alternative relations but still representing the same configuration space. Additionally, we investigated a decision-oriented product line for a steel plant configuration (*CC-L2*) [5], the *Dopler* Tool Product Line and several feature models (*WebPortal* by M. Mendoca, *Graph* by Hong Mei, *EShop* by Sean Quan Lau) available on the S.P.L.O.T. website (Software Product Lines Online Tools website [2]). Key characteristics of those models are stated in Table 3 like the number of questions ($\#q$), the number of choices ($\#c$) in the model, and the number of relations ($\#r$) between questions in the model. In addition the number of literals and clauses needed after the transformation into CNF is stated. Note that the feature models were automatically converted into our own decision model (basically features are represented by questions with up to three answers: yes, no, irrelevant) to be used with our tool, hence the characteristics differ from those given on the S.P.L.O.T. website.

Table 3: Decision Models Used for Evaluation.

Model	$\#q$	$\#c$	$\#r$	*#literals*	*#clauses*
Dell1	28	147	111	137	2,127
Dell2	24	147	23	142	2,540
Dopler	14	48	8	51	274
CC-L2	59	137	20	135	257
WebPortal	42	113	31	113	253
Graph	29	70	24	70	163
EShop	286	703	147	703	1,440

5.1 Objectives and Questions

The objectives of this evaluation are to investigate the effects of different isolation strategies on user guidance in decision-making scenarios. Specific questions we answer are:

1. What is the difference among the isolation approaches in terms of incomplete reasoning (missing guidance)?
2. What is the difference among the isolation approaches in terms of incorrect reasoning (faulty guidance)?
3. How many questions have to be revisited using the different approaches?
4. How are the approaches handling multiple inconsistencies at the same time?
5. How do the different isolation approaches for tolerating inconsistencies in SAT-based reasoning scale?

5.2 Execution

As mentioned in section 4.2 in order to be able to evaluate the isolation approaches with respect to our objectives, we need to know how an inconsistency is going to be fixed. For that purpose we generated one thousand valid configurations for each model (without any inconsistencies), to have a statistical significant sample size. In each configuration we injected up to three defects, for the purpose of getting an idea how the isolation strategies are effected when multiple defects are present three were sufficient because more defects are treated in the same manner. We seeded the defects by randomly changing decisions of each configuration to cause inconsistencies and treating the original decisions as the fixes. We then simulated the decision-making involved. Since each configuration contained defects, the decision-making eventually encountered an inconsistency. Starting at this point the simulation was continued using the different isolation approaches described in Section 4.1, while the reasoning data was collected for each

[2]http://www.splot-research.org/

simulation. The approach works with defects injected at any stage from the very beginning to the end, however for meaningful observations on completeness and correctness it is not useful to inject defects at the beginning or end due to the following reasons: i) assuring that uninvolved assumptions are made before the inconsistency detection, in order to reveal differences between the isolation approaches and ii) potentially leaving assumptions to be made left after the isolation, in order to be able to measure the impact of the different isolation approaches onto the reasoning with sufficient data points. This is why we injected defects in the range $[0.2 * \#q, 0.8 * \#q]$. Furthermore our simulations used a random decision order, changing the order may change the time when an inconsistency is detected or how many contributors the inconsistency has for each individual simulation, but it does not have any effects on the described isolation strategies and the results presented next.

To put our results concerning incomplete and incorrect reasoning into perspective, we also calculated the hypothetical best and worst cases. The worst case is that no SAT-based automation is available while tolerating inconsistencies (i. e., because it fails due to UNSAT and no strategy is in place for isolation). This implies that the worst case is about answering each question without any reasoning in place that helps guide the user. The ideal case is to isolate the defects only. We of course knew for each configuration where the defects were, since we seeded them. Thus, we can think of our knowledge as the optimal isolation strategy (although it should be clear that in a non-experimental setting this extra information would not be available and the ideal is not computable). Having the ideal available is useful for understanding how far away the various isolation strategies are from the optimum. As was mentioned above, Disregard All, Skip, and HUMUS compute unique isolation solutions while MaxSAT typically computes a non-deterministic solution out of several alternatives. To account for the randomness of the MaxSAT isolation we thus investigated it from its normal case (the random selection of a solution) as well as from its worst case (the solution does not isolate the defects and thus causes the maximum harm in terms of incorrect reasoning).

The results present the comparison of the different isolation strategies with the ideal isolation simulation data starting at the point during the decision-making process the first inconsistency was discovered. Looking at our example from Table 1 this would mean starting the comparison at the fourth question and ignoring literals belonging to questions already answered (the grayed out areas in the table). For instance to compute the incomplete reasoning data, we counted the number of literals remaining (no assumptions were derived for them) after each question answered, excluding literals belonging to already answered questions. To compute the incorrect reasoning data we counted the number of assumptions made different from the ones of the ideal simulation data, again excluding literals belonging to already answered questions. To compute the revisitation sizes the number of assumptions isolated were counted for each approach. And last but not least the needed isolation times were measured during the simulations.

5.3 Results

To get an idea what the evaluation data looks like, Figure 3 shows absolute results of three individual runs to assess incomplete reasoning with the HUMUS isolation strategy for the Dell1 model. On the x-axis the number of questions answered is shown, while on the y-axis the number of unassigned literals remaining is depicted. While these individual runs look quite different, their characteristics are basically the same, which is reflected in the overall results. The meaning of the top curve for example is that the inconsistency was detected after question 16 was answered, at this point after the HUMUS isolation took place there were 103 unassigned literals left belonging to the remaining 12 questions that were not yet answered or looked at. After answering question number 17 through reasoning the number of unassigned literals was reduced to 41 and so on.

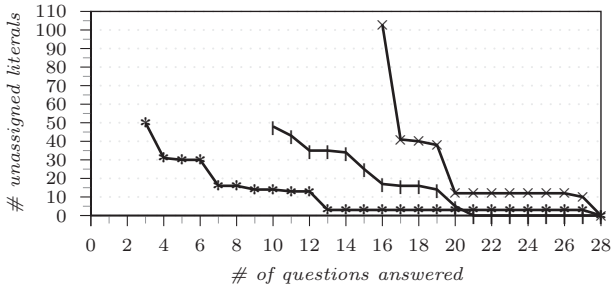

Figure 3: Incomplete Reasoning Progression Runs with HUMUS isolation strategy for the Dell1 Model.

For calculating the overall results covering all models, these individual runs were normalized on the x-axis between the question where the inconsistency was detected (0%) and the number of remaining questions in the model (100%) that need to tolerate the inconsistency. The y-axis values were normalized between the number of literals representing all the choices of the questions left at the time the inconsistency was detected (100% equal to no reasoning) and zero literals (0%). For example, the top curve from Figure 3, the point questions answered 17 and unassigned literals 41 will get normalized between 16 questions answered (0%) and 28 questions answered (100%) resulting in a x-value of $\sim$ 8.3%, the 41 unassigned literals will get normalized between 0 unassigned literals (0%) and 105 unassigned literals (100%), from the no reasoning simulation run, resulting in a y-value of $\sim$ 39.05%. Based on such normalized runs the averages shown in Figures 4 and 5 were calculated. Due to the very large number of configurations investigates, the averages have little variation and combined form a perfect line.

5.3.1 Incomplete Reasoning (Single Defect)

In Figure 4 the average results of all configurations evaluated for incomplete reasoning are shown (objective 5.1-1). The 95% confidence intervals are not shown to avoid further clutter, they are in the range of 0% up to 3.14% for all data points. Due to the fact that we only evaluate incomplete reasoning on questions not yet answered by the user, even with no reasoning 0% incomplete reasoning can be reached at the end when no questions are left. On the other hand even with knowledge that normally would not be available (ideal case) for about 30% of the remaining literals no assumptions can be derived at the time of the inconsistency detection. The results correspond to the general discussion beforehand in Section 4.2.1 but also hold some surprises. The Disregard All strategy which we expected to be the worst-case is in fact a good alternative to no reasoning at all and not just because it enables automations. The HUMUS isolation strategy on average only starts out with about 10% more incompleteness than the MaxSAT strategy

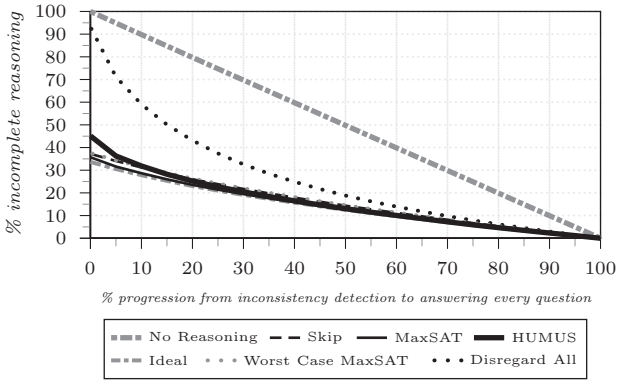

Figure 4: Incomplete reasoning progression results combining all case study systems.

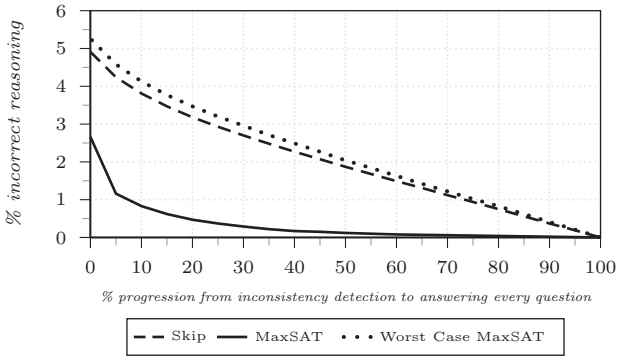

Figure 5: Incorrect reasoning progression results combining all case study systems.

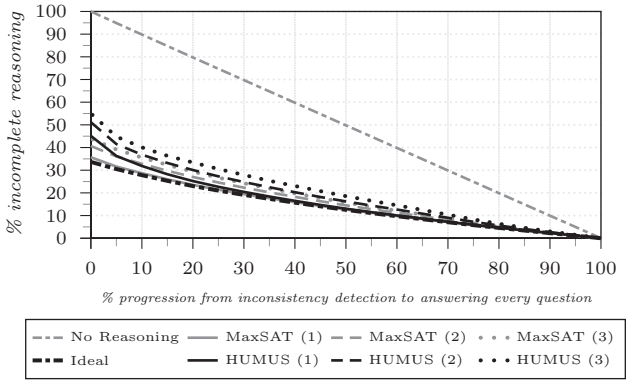

Figure 6: Incomplete reasoning progression results combining all case study systems with multiple defects.

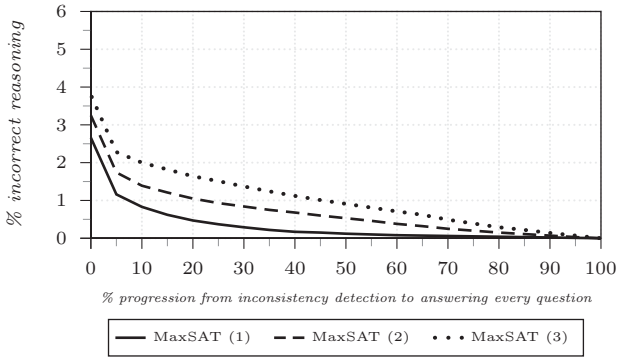

Figure 7: Incorrect reasoning progression results combining all case study systems with multiple defects.

and quickly closes the gap to the MaxSAT strategy (to the worst case MaxSAT at ∼10% and the random MaxSAT at ∼45% of the remaining configuration process). Overall it seems that the investigated models contain significant overlapping constraints resulting in a more complete reasoning than expected.

5.3.2 Incorrect Reasoning (Single Defect)

In Figure 5 the average results of all configurations evaluated for incorrect reasoning are shown (objective 5.1-2). Again the 95% confidence intervals are not shown to avoid further clutter, they are in the range of 0% up to 0.36% for all data points. The results for incorrect reasoning are also quite interesting. Overall the amount of incorrect derived assumptions seems not very high – in the worst case only about 6%. Another interesting fact is the self correcting ability of the MaxSAT strategy since reasoning with the defect sometimes leads to its re-detection in form of new inconsistencies due to the overlapping constraints. Every time such a defect is re-detected, the MaxSAT strategy has a chance to isolate the defect, increasing its chance over time and resulting in a rapid decrease in incorrect reasoning. The worst case MaxSAT obviously does not get that benefit and the Skip strategy also has no chance to eliminate the defect once it is included in the reasoning process since the defect must be located at an earlier point in time during the configuration. MaxSAT is thus an interesting alternative to HUMUS if incorrect reasoning is acceptable temporarily. However, do note that using MaxSAT in this manner is more equivalent to postponing inconsistencies since the defect is detected multiple times in form of different, yet related inconsistencies.

5.3.3 Revisitation (Single Defect)

The results show that the Disregard All isolated about 41% of the literals on average. The HUMUS isolation strategy (3,45%) on average isolates about twice as many literals as MaxSAT (1,27%), but less than Skip (3,94%) and the worst case MaxSAT (4,19%). This can be explained by additional isolations needed (increasing the number of literals in isolation) in case of additional inconsistencies, which occurred more frequent in the worst case (objective 5.1-3).

5.3.4 Multiple Defects

As mentioned earlier we also conducted simulations with up to three defects (objective 5.1-4). To avoid additional clutter only the HUMUS and MaxSAT strategies are shown in the figures. As evident in Figure 6 (95% confidence intervals are in the range of 0% up to 1.33% for all data points) the results for incomplete reasoning look quite the same except that naturally it increases for all strategies, since more defects mean more isolations, the same is true for revisitation results. However the results for incorrect reasoning in Figure 7 (95% confidence intervals are in the range of 0% up to 3,01% for all data points), in addition to a slight overall increase, clearly indicate a slower decrease in incorrect reasoning over time. This effect can be explained by the nature of MaxSAT to only isolate as little as possible and inconsistencies that involve common correct assumptions. Given our example of the three incompatible de-

Table 4: Scalability test results on artificial SAT problems.

#Contributors	*10*	*100*	*1000*	*10000*	*100000*
MaxSAT	1ms	1ms	3ms	86ms	∼6s
HUMUS	1ms	3ms	241ms	∼28s	∼1h

cisions $Screen\,Size(12.1'')$, $Screen\,Resolution(XGA)$ and $Webcam(yes)$, if two of them were injected defects then MaxSAT would always isolate the third correct one, until additional decisions conflict with the defects and as a result solutions with one decision isolated would not work anymore.

5.3.5 Scalability

We also conducted performance tests on an Intel® Core™ 2 Quad Q9550 @2.83 GHz with 4GB RAM, although only one core was used for the time being. The computation time needed for all models and the different isolation approaches was between 0ms and 1ms per computation. The evaluated models are not the largest SAT models around, however in context of this domain they are quite large and we have shown that the approach scales for our case studies (objective 5.1-5). However further evaluations on artificial SAT models (Table 4, a detailed description of how these models were created can be found in [17]) show an exponential growth but acceptable performance for inconsistencies involving up to 10000 assumptions. As mentioned in Section 5.3.3, the number of contributors (#Contributors), which is equal to the number of assumptions isolated by the HUMUS strategy averages around 3.45% of the model size for our case studies. Obviously the amount of contributors differs depending on the model size, the number and complexity of constraints and the inconsistency. However, that means in an interactive environment, HUMUS should be calculated fast enough for inconsistencies involving up to 1000 contributors which, given our observations, would put the model size at approximately 29000 features / choices.

5.4 Implications for Decision-Making

All four strategies are useful, because they allow automations to continue working in the presence of inconsistencies. The Disregard All and Skip strategy can easily be implemented for any reasoning engine and the results may be good enough in certain scenarios. The more advanced isolation strategies MaxSAT and HUMUS, are both viable options for tolerating inconsistencies although, at least for this domain, HUMUS appears superior to MaxSAT given that it avoids incorrect reasoning altogether (which we believe often to be worse than incomplete reasoning) and that its incomplete reasoning and revisitation effort is only briefly worse than that of MaxSAT. It also allows decision-making tools to visualize all contributing decisions of an inconsistency, which could help users to make an informed decision on how to fix an inconsistency, which is in our opinion superior to more or less random suggestions.

5.5 Threats to Validity

This work was evaluated on a diverse set of decision models. Still, there are threats to the validity which are discussed next.

Threats to construct validity imply whether we are evaluating the different strategies with the proper criteria. This paper evaluated the different trade-offs of common SAT-based strategies to tolerating inconsistencies in the domain of decision-making based on qualitative criteria (incomplete, incorrect reasoning, and revisitation) and other criteria (performance). While we acknowledge that these criteria may not be all there are, they seemed reasonable enough for our needs.

We made no assumption that would invalidate the internal validity of our findings. As was discussed, our approach applies to guided decision-making and was evaluated on predefinable decision models only. Fortunately, many decision models fall into this category and future work will show whether these strategies are applicable to tolerating inconsistencies in general.

Regarding external validity the question is: Are the case studies used for validating our approach representative of decision models in practice? Due to the fact that the case studies are from very different domains and real world examples, we can assume them to be representative with regards to composition and observable behavior. It seems reasonable that in other models relations will be as least as complex as in our case studies. We exhaustively evaluated the seven case study models by randomly injecting defects and observing the progression with regard to the different isolation approaches. Due to this exhaustive evaluation and the conclusions, we believe the conclusion validity to be high.

However, we cannot generalize that our result will apply to other domains where SAT-based reasoning is used. This paper thus provides a proof of concept in that we found domains where tolerating inconsistencies is indeed a viable option. We believe that many other domains would likewise benefit from observations we made here, though perhaps not all.

6. RELATED WORK

In this section we give a brief overview of work that already has been done in related research areas. The idea of tolerating inconsistencies is not new, 20 years ago, Balzer argued that inconsistencies should be detected and communicated to the developers; however, developers should not be hindered in continuing their work despite the presence of inconsistencies [1]. This basic but essential principle has been applied not only in the modeling world but for example also in any code editor that allows you to continue programming even if there is a syntax error [11].

SAT solvers and theorem provers [4, 3] are commonly used today. Concepts such as MUSes are well known in the SAT community [23], as are MaxSAT [12], MSSes, MCSes [14] and the CAMUS [13], but the application of these concepts in other software engineering domains, particularly for tolerating inconsistencies, has not been exploited as of yet.

As stated earlier in Section 2.3 SAT-based reasoning is state-of-the-art [15, 22, 16] in the domain of product line configuration. It has several primary uses: *i)* SAT reasoning is used to validate products, *ii)* find viable alternative solutions if a product is not valid [24], or auto complete partial products [15], and *iii)* to provide guidance during the configuration process [20, 19].

Other domains applying SAT-based reasoning like for instance hardware verification [9], debugging [21] and model checking [2, 10] already make use of the concept of MUSes to identify problems in the models. Even in non SAT-based reasoning environments like for example in UML modeling tools similar concepts are realized as a basis for fix generations [7, 8].

But to the best of our knowledge no one compared different isolation strategies and their trade-offs for tolerating inconsistencies as we have done in this paper.

7. CONCLUSION AND FUTURE WORK

This paper demonstrated four strategies for tolerating inconsistencies during SAT-based reasoning to guide users during decision-making and highlighted their respective advantages and disadvantages. While we only evaluated this one SAT-based automation of guiding the user during decision-making, it is a proof of concept, that tolerating inconsistencies during SAT-based reasoning is possible, and even that incorrect reasoning is avoidable, without the need to adapt the automation itself. It is our belief that the same properties and observations should also hold for other SAT-based automations in decision-making, and even for SAT-based automations in other domains. We come to this conclusion because of the fact that these strategies do not have to understand any details about the automation to function, in fact they are applied to the SAT reasoning itself. This implies that the strategies can be readily evaluated and adopted in other domains. As a next step we think it is important to also consider fixing defects which we believe to be tightly related to the tolerating of inconsistencies.

Acknowledgments

The work was kindly supported by by the Austrian Science Fund (FWF): P21321-N15 and the Austrian Center of Competence in Mechatronics (ACCM), a K2-Center of the COMET/K2 program, which is aided by funds of the Austrian Republic and the Provincial Government of Upper Austria. The authors thank all involved partners for their support.

8. REFERENCES

[1] R. Balzer. Tolerating Inconsistency. In *13th ICSE, Austin, Texas, USA*, pages 158–165, 1991.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *DAC*, pages 317–320, 1999.

[3] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[4] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[5] D. Dhungana, P. Grünbacher, and R. Rabiser. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, 18:77–114, 2011.

[6] A. Egyed. Instant consistency checking for the UML. In *28th ICSE, Shanghai, China*, pages 381–390, 2006.

[7] A. Egyed. Fixing Inconsistencies in UML Design Models. In *29th ICSE, Minneapolis, USA*, pages 292–301, 2007.

[8] A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *23rd ASE, L'Aquila, Italy*, pages 99–108, 2008.

[9] A. Gupta, M. Ganai, and C. Wang. SAT-Based Verification Methods and Applications in Hardware Verification. In M. Bernardo and A. Cimatti, editors, *Formal Methods for Hardware Verification*, volume 3965 of *Lecture Notes in Computer Science*, pages 108–143. Springer Berlin / Heidelberg, 2006.

[10] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000.

[11] C. W. Johnson and C. Runciman. Semantic Errors - Diagnosis and Repair. In *SIGPLAN Symposium on Compiler Construction*, pages 88–97, 1982.

[12] C. M. Li and F. Manyà. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, pages 613–631. 2009.

[13] M. H. Liffiton and K. A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *J. Autom. Reason.*, 40(1):1–33, 2008.

[14] M. H. Liffiton and K. A. Sakallah. Generalizing Core-Guided Max-SAT. In *Theory and Applications of Satisfiability Testing - SAT, 12th International Conference, Swansea, UK*, pages 481–494, 2009.

[15] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: software product lines online tools. In *24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, New York, NY, USA, 2009. ACM.

[16] M. Mendonça, A. Wasowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *Software Product Lines, 13th International Conference, San Francisco, California, USA*, pages 231–240, 2009.

[17] A. Nöhrer, A. Biere, and A. Egyed. Managing SAT inconsistencies with HUMUS. In U. W. Eisenecker, S. Apel, and S. Gnesi, editors, *VaMoS*, pages 83–91. ACM, 2012.

[18] A. Nöhrer and A. Egyed. C2O: A Tool for Guided Decision-making. In *25th International Conference on Automated Software Engineering, Antwerp, Belgium*, pages 363–364, 2010.

[19] A. Nöhrer and A. Egyed. Optimizing User Guidance during Decision-Making. In *Software Product Lines, 15th International Conference, Munich, Germany*, 2011.

[20] M. L. Rosa, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Questionnaire-based variability modeling for system configuration. *Software and System Modeling*, 8(2):251–274, 2009.

[21] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(10):1606–1621, 2005.

[22] P. Trinidad, D. Benavides, A. R. Cortés, S. Segura, and A. Jimenez. FAMA Framework. In *Software Product Lines, 12th International Conference, Limerick, Ireland*, page 359, 2008.

[23] H. van Maaren and S. Wieringa. Finding Guaranteed MUSes Fast. In *Theory and Applications of Satisfiability Testing, 11th International Conference, Guangzhou, China*, pages 291–304, 2008.

[24] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. R. Cortés. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Software Product Lines, 12th International Conference, Limerick, Ireland*, pages 225–234, 2008.