Managing SAT Inconsistencies with HUMUS

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

In Product Line Engineering, as in any other modeling domain, designers and end users are prone to making inconsistent assumptions (errors) because of complexity and lack of system knowledge. We previously envisioned a way of allowing inconsistencies during product configuration and in this paper we present a solution on how to realize this vision. We introduce HUMUS (High-level Union of Minimal Unsatisfiable Sets), which enables correct reasoning in product line engineering (encoded in SAT) despite the presence of errors. We focus mainly on tolerating inconsistencies during product configuration, to make it possible to resolve inconsistencies later without misguiding the human user along the way. We also provide a discussion of other applications in product line engineering and beyond. The main advantage of using HUMUS is, that it is possible to isolate erroneous parts of a product line model such that existing automations continue to be useful. The applications of HUMUS are thus likely beyond product line engineering.

Categories and Subject Descriptors: I.6.4 Simulation and Modeling: Model Validation and Analysis

General Terms: Algorithms, Human Factors, Verification.

Keywords: Product Line Engineering, Formal Reasoning, User Guidance

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. Almost 20 years ago, Balzer wrote that "software systems, especially large ones, are rarely consistent (...) yet no principled basis exists for managing the development during the periods of inconsistency". He 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.

In model-driven software engineering, it is state-of-the-practice to allow inconsistencies [1, 21]. Modeling tools tend to indicate inconsistencies but do not force developers to fix them right away [10]. However, in the product line community and many other software engineering domains, tolerating inconsistencies is usually infeasible (e. g., by preventing decisions that cause inconsistencies). And there are good reasons for preventing inconsistencies. First and foremost, many reasoning engines are rendered partially or fully useless in the presence of inconsistencies. Even if the reasoning engines were able to function (instead of failing outright), the implications on the quality of the results are typically not understood (clearly, 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 support many vital automations such as analyzing properties of systems, understanding the effects of design decisions, helping configure products, etc.

The goal of this work is to investigate the cause of inconsistencies with the help of SAT-based reasoning [6], more specifically HUMUS (High-level Union of Minimal Unsatisfiable Sets, which will be explained in detail in the following section). We will highlight advantages and disadvantages of conservatively identifying the cause of inconsistencies and propose several usage scenarios in which doing so benefits product line engineering. This paper thus continues on the vision in [22], where we discussed different inconsistency resolution strategies: from undoing, forced repairs, to the tolerance of inconsistencies by ignoring all facts that contribute directly or indirectly to the inconsistency. This enables the designer to better understand inconsistencies and resolve them at a time of their choosing. We will demonstrate that *i)* correct reasoning in the presence of inconsistencies is possible and *ii)* automations (SAT-based) remain useful even while tolerating inconsistencies. We thus counter the assertion above: a user can expect a reasoning engine to compute correct results for as long as the incorrect input is detected and isolated during the reasoning (albeit not resolved).

The focus on SAT-based reasoning reflects the wide-spread use of this technology in product line engineering. SAT-based reasoning is used to determine the correctness of product line models and to help guide decision makers – for example, revealing which decisions / features are no longer available due to conflicts with decisions / features made / selected

VaMoS'12, January 25–27, 2012, Leipzig, Germany.
Copyright 2012 ACM 978-1-4503-1058-1/12/01 ...$10.00.

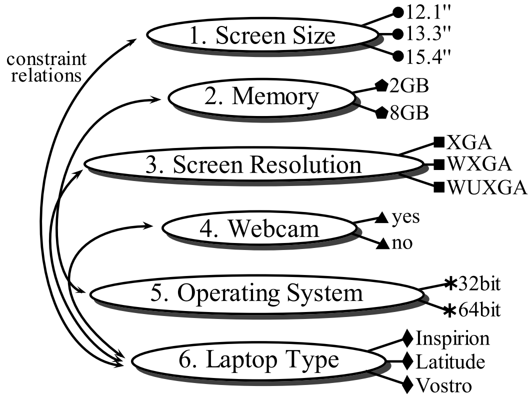

Figure 1: Illustrative Excerpt of a Product Line Decision Model.

earlier. One goal is to investigate how such and other useful automations, that are relying on SAT-based reasoning, can continue to be useful even if the SAT model itself contains errors. Particularly for SAT-based reasoning, this is a severe problem because a SAT solver renders any model with an inconsistency as "not satisfiable" regardless of how minuscule or irrelevant the inconsistency may be. Tolerating inconsistencies thus strips the engineer of vital automations because the SAT solver ceases to compute useful results.

This paper thus contributes a SAT-based strategy to conservatively identifying the cause of inconsistencies and demonstrates interesting usage scenarios. While the evaluation in this paper mainly focuses on guidance during product derivation, HUMUS can be applied to all kinds of SAT-based reasoning.

Next, we present an illustration used throughout the paper. We discuss the goals of this work and the challenges. A detailed discussion of the usage scenarios follows. The preliminary evaluation then examines four case studies, followed by a discussion of the results. Related work, future work and conclusions round out the paper.

2. SCENARIO AND PROBLEM

Since this paper investigates the HUMUS (High-level Union of Minimal Unsatisfiable Sets) strategy to identify the cause of inconsistencies, which is specific to *SAT solvers*, a short introduction to the terminology based on [5] 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. 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 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:

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

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

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

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

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 SAT-based Reasoning in Product Lines

In the domain of product configuration and decision modeling, SAT-based reasoning is state-of-the-practice [19, 26]. It has several primary uses: SAT reasoning is used *i)* to validate products [3], *ii)* to find viable alternative solutions if a product is not valid [28] or auto complete partial products [19], and *iii)* to provide guidance during the configuration process [24, 23]. 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.

But what happens if we are confronted with inconsistencies in the product line model or during the product configuration? 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 before, since the SAT solver is used for more then just detecting inconsistencies, automations are lost too. For example, the ability to automatically

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

derive assumptions *i)* to (partially) auto-complete the configuration process or *ii)* show decision effects, is lost. On the other hand if we are dealing with an inconsistent product line model the ability to identify for example dead features or to do a partial configuration is also lost.

If the tolerance to inconsistencies is a necessity, there are two possibilities for maintaining automations: either *i)* change the SAT-based automations to live with inconsistencies (this is difficult in part because there are many different kinds of automations), or *ii)* change the SAT input (clauses and assumptions) so that the problem is no longer unsatisfiable which implies that existing automations continue working without modifications. For changing the SAT input our preferred way 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 product lines, 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). Hence our focus on high level clauses.

3. APPROACH

In this section we will discuss the HUMUS strategy (High-level Union of Minimal Unsatisfiable Sets) to identify the cause of inconsistencies in detail. To provide a better insight into this SAT technique we need to introduce a few more SAT concepts [5, 17]. The most important being a minimal unsatisfiable set (MUS) which is defined by the properties of being minimal and that removing any single assumption / clause 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 / clause 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 high-level (assumption based) MCSes are $\{Screen\,Size(12.1'')\}$, $\{Webcam(yes)\}$, and $\{Screen\,Resolution(XGA)\}$. 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 maximum 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 $\{Memory(8GB), Screen\,Resolution(XGA), Webcam(yes)\}$.

As stated before HUMUS stands for High-level Union of Minimal Unsatisfiable Sets, it is a concept based on the calculation of all Minimal Unsatisfiable Sets (MUSes) [17]. The basic concept behind it is to identify all contributors (directly and indirectly) of the inconsistency. When we speak of HUMUS we are always talking about identifying high-level assumptions, UMUS (Union of Minimal Unsatisfiable Sets) on the other hand identifies low-level clauses that are contributing to an inconsistency that cannot be resolved with the removal of assumptions, only with the removal of clauses. Note that the HUMUS / UMUS calculation only returns a single result, because an assumption / clause either contributes to the inconsistency (in which case it is in the HUMUS / UMUS) or it does not.

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

The implementation of the HUMUS strategy takes a shortcut in comparison to the approach of Liffiton to compute all MUSes [17], since we only care about the union of the MUSes and not the individual MUSes themselves. Our implementation uses a variant of Liffiton's [17] approach to calculate MSSes using assumptions over clause selector variables (since clauses in CNF are disjunctions adding a variable to be used as a selector is easy, if the clause should be ignored true is assumed for the variable which results in the clause being true). 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 (the negation of the set as added to the SAT model as a clause, eliminating the particular set as a satisfiable assignment). 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.

4. USAGE SCENARIOS

HUMUS or its low-level counterpart UMUS can be useful in many application scenarios, of which we will explain a few we could think of in more detail in this section.

4.1 Product Configuration

SAT-based reasoning can be used to provide basic guidance, by calculating the effect of a decision and show its effects to the user when subsequent 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 $\neg Laptop\,Type(Vostro)$). This process is repeated after each decision during the whole process.

In the illustration in Table 1 technically after the third decision the configuration 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 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.

Our vision was to allow users to continue the configuration process while tolerating the inconsistency [22]. Calculating the HUMUS and isolating the identified decisions for the time being allows us to do exactly that, however it is also able to support the other two possibilities in such a situation which is explained in the following subsections.

4.1.1 Inconsistency Explanation

In case one wants to disallow conflicts, one may still want to support users by providing an explanation to them, why a certain decision is not allowed. If the intend is to provide a full explanation including all transitive effects calculating the HUMUS is the answer. In the example given in our illustration the HUMUS consists of $\{Screen\,Size(12.1'')$, $Screen\,Resolution(XGA)$, $Webcam(yes)\}$, so if the system is asked why $Webcam(yes)$ is not allowed, the answer would be because of the conflicting decisions $Screen\,Size(12.1'')$ and $Screen\,Resolution(XGA)$.

4.1.2 Fixing Inconsistencies Right Away

In case one prefers to fix inconsistencies right away, HUMUS provides the basis for all possible fixes. Since all contributing decisions are included in the HUMUS, every possible fix has to involve changing one or more of those decisions. So if the intend is not only to suggest randomly selected fixes but a complete list of fixes to the users, it can be done by starting to calculate the HUMUS to identify the contributing decisions and finding alternative ones like for example $Screen\,Resolution(WXGA)$, $Screen\,Size(13.3'')$ or $Webcam(no)$ for our illustration.

4.1.3 Tolerating Inconsistencies

If an explanation or selecting a possible fix is not sufficient enough for the user to decide on a certain fix, tolerating can be a useful alternative. As mentioned before once an inconsistent state is reached and the SAT-Solver returns UNSAT, automations stop working. As a result tolerating inconsistencies with SAT-Solvers requires to isolate clauses and / or assumptions from reasoning that led to the UNSAT result. Of course isolating any number of contributors of an inconsistency could be sufficient, to get meaningful results out of the SAT-Solver again, like for instance $Webcam(yes)$. However it is our belief that isolating conservatively all contributors, as calculated by the HUMUS, has one big advantage. Isolating all contributors ensures that the decision that is changed later on by the user to fix the inconsistency is not used for reasoning, and therefore not used in any automations. Not doing so could lead for instance to valid decisions (based on the later known fix) being communicated to the user as invalid during the configuration process, based on reasoning with the defective decision. For example if the user would decide later on that the decision $Screen\,Size(12.1'')$ should be replaced with $Screen\,Size(13.3'')$, but we just isolated $Webcam(yes)$ from reasoning, $Screen\,Size(12.1'')$ would have been part of the reasoning and led to incorrect reasoning results like for instance $\neg Laptop\,Type(Vostro)$. So isolating all contributors eliminates the possibility of incorrect reasoning and ensures correct automations.

4.1.4 Fixing through Tolerating Inconsistencies

Tolerating can even be beneficial for fixing an inconsistency if new decisions can be trusted, after the inconsistency was detected. If new decisions have dependencies with decisions contained in the HUMUS they can be used to reduce the number of possible fixes for the inconsistency. Sometimes the number of possible fixes can even be reduced to one, in which case the inconsistency can be fixed automatically. For instance given our example if the user makes a decision for the $Laptop\,Type$ there are three possible choices $\{Vostro$, $Inspirion$, $Latitude\}$, where each choice resolves the inconsistency in a different way. If $Vostro$ is chosen, and can be trusted to be correct, there exists an additional conflict with $Screen\,Size(12.1'')$, which means that this decision needs to be changed and the decisions $Screen\,Resolution(XGA)$ and $Webcam(yes)$ can be kept. On the other hand if $Inspirion$ is chosen there is an additional conflict with $Screen\,Resolution(XGA)$ and as a result $Webcam(yes)$ and $Screen\,Size(12.1'')$ can be kept. The same is true for $Latitude$ which is in conflict with the decision $Webcam(yes)$.

The fixing possibilities could be further reduced, if the trusted decisions also included the decision that caused the inconsistency. As we already argued in [22] we assume that if a user consciously decides to introduce an inconsistency (because the configurator tool already indicated that the decision $Webcam(yes)$ is not allowed anymore), this last decision is important and should be kept.

But what if follow-on decisions also cannot be trusted to be correct? For instance because they are made at different times during the configuration process or by different stakeholders. For the tolerating part, HUMUS still allows automations to function correctly, even if new inconsistencies are detected a new HUMUS will be calculated without any incorrect reasoning happening. For the fixing part, if we cannot trust decisions made after the detection of an inconsistency (in our experience rarely the case) or if continuing the configuration does not reveal a single fix (more likely the case) then the user will have to fix the inconsistency by traditional means – for example, by generating fixes based on the HUMUS for the entire set of decisions and not just the decisions made prior to the inconsistency detection; or by calculating a more or less random fix. A straightforward way (we currently use) for generating fixes based on the HUMUS for decision models is to identify the questions the contributing decisions belong to. The next step is to test still viable alternative choices for those questions and combinations thereof for satisfiability; and collecting those that are valid as possible fixing sets. As mentioned before, new decisions made while tolerating inconsistencies, can be used to reduce those fixing sets if they conflict with those new decisions. As such this approach of reducing the number of possible fixing sets only works if new decisions can be trusted. By visually communicating the impact of new decisions onto fixing possibilities to the user and making those new decisions informed ones, one could argue that the assumption that one can trust those decisions is not to far fetched.

4.2 Product Line Verification

SAT solvers are also useful in performing product line verification. For example detecting anomalies [3] like for instance dead features and false optional features. However detecting such anomalies in an inconsistent SAT model is impossible because the SAT solver will return UNSAT for every query. In such cases the UMUS could be calculated and the identified clauses excluded from the analysis. Dead features and false optional features are then detected in the remaining partial product line model, which is more useful then failure to evaluate anything. And faulty features identified this way will remain faulty when the inconsistencies in the model are fixed. This statement is true because if certain constraints cause a feature to always or never be selected adding additional constraints (clauses) to the model will not change these effects. Additionally because the UMUS identified all contributors no potentially incorrect constraints are used to identify those dead and false optional features, as a result the used constraints to identify them will still be in the consistent final product line model. The applicability of HUMUS for these kinds of reasoning is only then useful if reasoning over a partial model provides correct results. In other cases, reasoning with HUMUS may be an approximation and the usefulness depends on other factors not explored here.

4.3 Other Domains

While we only investigated the benefits of HUMUS in the product line configuration scenario (see the next section), we strongly believe that the basic premise also applies to other domains. For instance SAT solvers are commonly used for debugging purposes [25], by identifying MUSes. But we believe that also in the domain of debugging it is far more beneficial to provide engineers with a complete list of contributors, rather than random ones derived by other techniques which are commonly used in state of the art. In order to understand a bug completely and allow engineers to make informed decisions on how to fix a bug all contributing parts have to be identified, in addition as mentioned in Section 3 identifying one single MUS and changing one element of it is not always enough to resolve an inconsistency.

Generally speaking in situations where engineers have to resolve or to live with any kind of inconsistency, it is our believe that identifying the complete set of contributors is the preferable way of dealing with inconsistencies and allows automations to function in a correct way without misleading engineers.

Table 2: Models used for the preliminary evaluation.

Model	*#q*	*#c*	*#r*	*#literals*	*#clauses*
Dell	28	147	111	137	2127
V1	59	137	20	135	257
WebPortal	42	113	31	113	253
Graph	29	70	24	70	163

5. PRELIMINARY EVALUATION

HUMUS is useful if an inconsistency only partially affects the SAT-based reasoning (automation). In this case, HUMUS isolates the problematic part conservatively and enables existing SAT-based automations to function again and provide useful results based on reasoning with the remaining consistent parts. However, one question remains, namely are the automations with the remaining parts still useful? This section demonstrates that in context of product line configuration, conflicts typically only require a partial model to be isolated and hence there is a benefit in correctly and completely identifying this problematic part with HUMUS. While this evaluation does not cover all described usage scenarios (see Section 4), it can be seen as a proof of concept. Future work will expand this evaluation onto other areas.

In this preliminary evaluation we investigated several aspects of HUMUS utilized during product configuration specific to user guidance, where guidance is about disabling choices for future questions based on decisions made. We investigated several 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 *DELL* 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. Due to the number of relations per question, this model was also the most complex decision model, as is reflected by the number of clauses. Additionally, we investigated a decision-oriented product line for a steel plant configuration (*V1*) [7] and two feature models (*WebPortal* by M. Mendoca, *Graph* by Hong Mei) 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 2 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 auto-

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

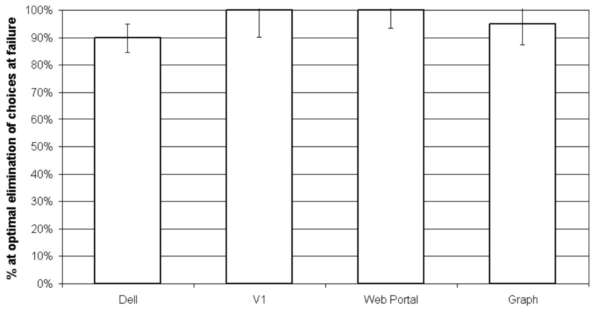

Figure 2: Effectiveness of guidance despite conservative isolation.

matically 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.

In order to be able to evaluate the HUMUS approach, 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 statistically significant sample size. In each configuration we seeded a defect by randomly changing a decision of each configuration to cause an inconsistency and treating the original decision as the fix. We then simulated the decision-making involved. Since each configuration contained a defect, the decision-making eventually encountered an inconsistency. Starting at this point reasoning data was collected and later analyzed.

5.1 Tolerating Inconsistencies during Product Configuration

Our approach is conservative in always isolating the defect, however, at the expense of also potentially isolating correct decisions. This reduces the effectiveness of automated guidance. Recall that the aim of guidance is to disable choices of questions that are no longer allowed. The conservative nature of our approach does so less effectively. Figure 2 measures this disadvantage. The data was normalized such that 100% on the y-axis represents optimal guidance where only the defect is isolated from reasoning (ideal); and 0% represents the worst case guidance where all decisions are isolated and the reasoning process starts over. We see that guidance remains 90-100% optimal despite the conservative nature of HUMUS due to two factors we identified. One being that, if the constraints in the model have a lot of overlaps and the number of decisions identified by HUMUS is large, lost information can be replaced with new decisions due to other constraints. The other being that, if the constraints in the model have little overlaps, then the number of decisions identified by HUMUS is small.

5.2 Fixing through Tolerating Inconsistencies during Product Configuration

To evaluate the fixing aspect of tolerating inconsistencies, we investigated how many inconsistencies can be fixed automatically. And in cases where this is not possible, how many fixing possibilities can be excluded over time.

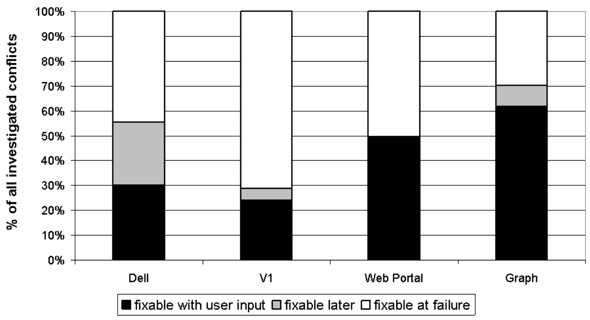

Figure 3: Distribution of fixable situations without additional user interaction.

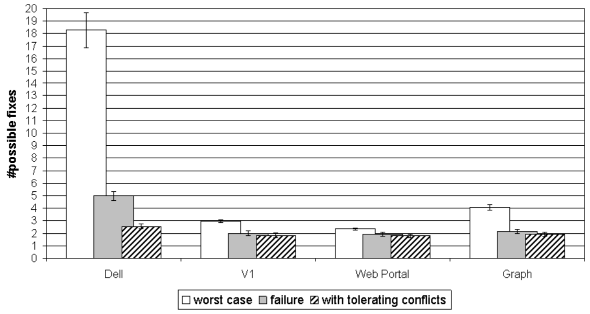

Figure 4: Overview over the number of possible fixes.

5.2.1 *Automatically Fixing*

At the time of the failure, three situations are possible:

1. A single fix is already computable when the inconsistency is detected and the decision that caused the inconsistency is trusted to be correct (*fixable at failure*).

2. A single fix is not computable when the inconsistency is detected but becomes computable at some point if follow-on decisions can be trusted (*fixable later*).

3. A single fix is not computable even at the end, with follow-on decisions trusted, however, the number of choices are reduced making it easier to fix the defect (*fixable with user input*).

We can see in Figure 3, that 29-71% of defects were fixable at the time the inconsistency was detected (i. e., there is only one option available and fixing it is trivial). The remaining defects required more user input. However, 0-25% of the remaining defects were fixable simply by letting the decision-making continue (while the inconsistency was tolerated) without requiring additional information. Tolerating inconsistencies thus fixes defects automatically in many situations. Even in the cases where defects were not fixed automatically, the choices for fixing them got reduced considerably during tolerating. This benefit is discussed next. This demonstrates that the fixing of defect is not trivial in many cases.

5.2.2 *Automatically Reducing Choices for Fixing*

For the 29-71% of defects in Figure 3 that were not fixable at the time the inconsistency was detected, Figure 4

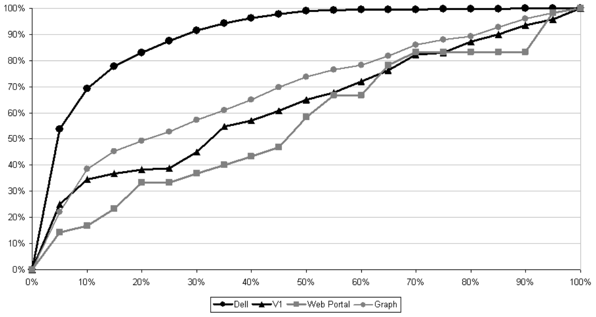

Figure 5: Normalized progression of fix reduction.

presents the actual number of possible fixes at the time of the detection and at the end (with their respective confidence intervals of 95%). Three situations are distinguished here:

1. The number of possible fixes if the HUMUS is computed at the end after all decisions have been answered (*worst case*).

2. The number of possible fixes if the HUMUS is computed when the inconsistency is detected (*failure*).

3. The number of possible fixes if the HUMUS is computed when the inconsistency is detected, but further reduced by using follow-on decisions with the assumption that they can be trusted (*with tolerating conflicts*).

The first situation does not distinguish between decisions made prior to and after the detection of the inconsistency. Follow-on decisions are not trusted to be correct and thus there are many choices for fixing the inconsistency. The second situation recognizes that the defect must be embedded among the decisions made prior to the detection. This simple knowledge vastly reduces the number of possible fixes. Tolerating the inconsistency then further improves on this by considering the effect of decisions made after the detection (i. e., while tolerating inconsistencies) onto the decisions made prior (the optimal is '1'). Tolerating inconsistencies thus makes it easier to fix defects. The improvements observed in Figure 4 are more substantial in decision models where there are the more relations (e. g., DELL model). This is easily explained. The more relations (constraints) there are in a model, the more knowledge can be inferred and thus the more restricted are the number of possible fixes.

In Figure 5 the progression of this improvement is shown relative to the percentage of decisions remaining until the end. We see that it is not always necessary to continue decision-making until the end to get the most out of tolerating inconsistencies. On the decision model with the most relations (DELL), we observe that 50% of the remaining questions answered after the detection (while tolerating inconsistencies) are enough to achieve near optimal reasoning. If the decision model is less constrained, the effect appears more linear. However, we observe that every decision made after the detection simplifies the fixing of the defect. Note that the 0% marker on the x-axis corresponds to the point of the detection and the 100% marker to the end of decision-making. The y-axis denotes the percentage of choices reduced for fixing defects compared to the optimum which is equal the number of choices reduced at the end (once all user input is known). Note that we excluded all those cases where no more reduction was possible after the detection (this data would be always at 100% and therefore distort the other results).

Table 3: Scalability test results on artificial SAT problems.

#Contributors	*10*	*100*	*1000*	*10000*	*100000*
HUMUS	1ms	3ms	241ms	27s 608ms	1h 897ms

5.3 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 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. However further evaluations on artificial SAT models (Table 3) show an exponential growth but acceptable performance for inconsistencies involving up to 10000 assumptions. While those artificial SAT models may not represent the structure of typical decision models well, they represent the worst case structure for our implementation. Those artificial SAT models consist of a single clause containing n literals ($l_1 \vee l_2 \vee \ldots \vee l_n$) and then all literals are assumed to be set to *false* resulting in an inconsistency because at least one literal has to be set to *true*. While this kind of SAT model may seem trivial, our HUMUS implementation determines all MCSes by searching for all MSSes and taking the complementary set of each one. As a result the number of MSSes grows linearly ($n - 1$, where n is the cardinality of the elements in the clause), but so does the number of elements in each MSS. Our current implementation builds a single MSS bottom-up which means there are $n!$ SAT calls necessary, to determine one MSS.

5.4 Discussion

This preliminary evaluation, although only configurations with a single defect were analyzed, already provides promising result to use HUMUS in user guidance during product configuration. It clearly shows that the effects on the reasoning are marginal and tolerating an inconsistency is not only possible with an SAT solver, but can even be beneficial for fixing it later. It also shows that calculating the HUMUS scales quite well for reasonable HUMUS sizes.

Of course there are several threads to validity and open issues, since it is only a preliminary evaluation. First of all, not all described usage scenarios have been tested, since our research interests are mostly focused on guidance and freedom during the configuration process itself. On the other hand we wanted to show in this paper, that the usage of HUMUS seems feasible in other areas too, even though we cannot substantiate that with more than arguments at this point. The preliminary evaluation on tolerating inconsistencies although not thorough yet, hints at the feasibility of our approach. We also realize that for fixing inconsistencies through tolerating, our assumption that new decisions can be trusted is not always realistic , especially considering the fact that an inconsistency may be the result of multiple defects. However, it provides us with results under optimal conditions and just by remembering the point of failure in combination with HUMUS, the number of possible fixes can be significantly reduced compared to searching for a fix at the end of the configuration as shown in Figure 4. This always works even if new decisions cannot be trusted and need additional fixing. And last but not least, it is still unknown how users would use this approach to manage inconsistencies. It is our believe that tolerating inconsistencies is helpful but user studies are needed to confirm this.

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 [15].

In software engineering, typical applications for decision-making are software installation wizards that guide the users through a set of predefined choices on how to setup a software system; product configuration and product line engineering makes use of models to restrict how products may be instantiated [8]; feature location and other forms of traceability [9]. Guided decision-making is quite common whenever software engineers desire to restrict the space of possible answers (e. g., installation wizards, e-commerce, product lines or process configurations).

SAT solvers and theorem provers [6, 5] are commonly used today. Concepts such as MUSes are well known in the SAT community [27], as are MaxSAT [16], MSSes, MCSes [18] and the CAMUS [17], 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 2.2 SAT-based reasoning is state-of-the-art [19, 26, 20] in the domain of product line configuration. The translation of configuration problems / feature models / decision to CSPs is solved and described for example in [2]. It has several primary uses: *i)* SAT reasoning is used to validate products [3], *ii)* find viable alternative solutions if a product is not valid [28], or auto complete partial products [19], and *iii)* to provide guidance during the configuration process [24, 23].

Other domains applying SAT-based reasoning like for instance hardware verification [13], debugging [25] and model checking [4, 14] 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 [11, 12].

To the best of our knowledge no one has used the union of MUSes for the purpose of better understanding inconsistencies and dealing with them before, as we have done in this paper.

7. CONCLUSIONS AND FUTURE WORK

This paper investigated the HUMUS SAT approach for conservatively identifying the cause of inconsistencies. While using the union of MUSes is not complicated and based on well-known SAT concepts, it has never been used in applications we propose and is not very well-known outside the SAT community. So our intention is to raise the awareness for this technology and its benefits for dealing with inconsistencies. We also provided several usage examples and evaluated the application of HUMUS in the product configuration scenario.

Open issues that also need to be investigated are how HUMUS performs dealing with several independent inconsistencies during the configuration process and how it could be applied to a multi-user configurator. In addition to that, it will also be interesting to investigate if HUMUS is as useful as we think in the other proposed usage scenarios, or may even have a wider applicability and be beneficial to other areas in software engineering that rely on SAT-based reasoning, e. g. SAT-based hardware verification [13] or debugging [25].

Acknowledgments

The research was funded by the Austrian Science Fund (FWF): P21321-N15.

8. REFERENCES

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

[2] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated Reasoning on Feature Models. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE, Porto, Portugal*, pages 491–503, 2005.

[3] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[4] 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.

[5] 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.

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

[7] 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.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel. DOPLER: An Adaptable Tool Suite for Product Line Engineering. In *Software Product Lines, 11th International Conference, Kyoto, Japan*, pages 151–152, 2007.

[9] A. Egyed. Resolving uncertainties during trace analysis. In *12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Newport Beach, CA, USA*, pages 3–12, 2004.

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

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

[12] 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.

[13] 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.

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

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Trans. Internet Techn.*, 2(2):151–185, 2002.

[22] A. Nöhrer and A. Egyed. Conflict Resolution Strategies during Product Configuration. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria*, volume 37 of *ICB Research Report*, pages 107–114. Universität Duisburg-Essen, 2010.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.