Abstract. The importance of variant development structures has increased continuously over the past few years. Nowadays the keyword is mass customization. Manufacturers have to satisfy the personal needs of their clients to keep up with competitors. Individual wishes and increasing demands of customers require the possibility of flexible and nearly limitless adaptations of a product. The result is a diversity of variants in one product line. Issues occur during the development of the corresponding development structures that were related to the complexity of the arising product data. Not only the amount of functionality and therefore individual components is rising, but also the interrelationships among the single components are getting more complex. The number of new evolving variants once a feature is added increases in the worst case exponentially. The resulting complexity cannot be handled manually. Thus, a formal logic based approach has to be used to describe the underlying variability model of the product structure. These formal specifications provide a basis for algorithms, which analyse the structures in terms of finding all kinds of errors like inconsistencies or dead features. Such results include formal proofs, which reason about the derivation of the found errors. As the users who construct and manage the development structures typically have no expert knowledge about formal languages and proofs, the analysis output has to be represented in a role and user-specific way. The presented work concentrates on an approach to visualize the formal results in an understandable, adaptable and user-oriented fashion. Different concepts are elaborated, which cover the information needs of specific user groups to match their respective knowledge level. As feature models are used to represent variant development structures in a simple and compact manner, they are used as a basic visualization technique. Other views represent the proof, in fact a resolution graph, a proof tree and a proof step. One possibility to understand the proof is to simulate through the individual steps. Each of the features and relationships, which play a role in the current step, are highlighted in the feature model. The mapping between proof steps and features and their relationships simplifies the comprehension. Based on these concepts a prototype is implemented, whose functionality respects the common human computer interaction requirements. To conclude, the result is summarized and prospects on future increments, further concepts and possible improvements are given.

Keywords. Visualization, Proof, Variance, Resolution
Introduction

Variants are standard supply by now in almost every industry. Issues occur because of
the development structures which are constantly changing.

In former times one simple product consisted of a countable amount of single parts,
whose composition was fixed. Mass production did not tolerate any individualization
of a product. The formal specification was simple and of little scope. If there was any
problem with a newly introduced feature, the so-called local hero was able to solve it
with minimal effort, because he knew all of the specification. Nowadays as the industry
is focused on the customer, variant diversity increases continually and thus the
complexity of product specifications. If a new component is added to the development
structure, not only one new configuration evolves, but the number of different variants
explodes exponentially. It is not possible to preserve an overview of the specification
and not to mention to find inconsistencies or solve problems. Therefore automatic
analyses are needed, which discover inconsistencies or dead features. The output of
such analyses is a formal proof, which base on SAT instances, as the specification was
translated into formal language before. As several user groups with different levels of
knowledge and information demands may need to understand the reason of the
inconsistency, it is required to visualize the line of proof and thus the cause in an
adequate way for each of the user groups. The result of the analysis needs to be
processed, so that the user is able to understand the proof. Information is visualized,
adapted to his specific needs. The main aim is to support the user via arbitrary
interaction with the developed system, which means, he can choose from
predetermined modes of visualization to manual setting of the display. For a first
impression of the current state of research, two tools were found, which are described
in the next section.

1. Existing tools

Existing Tools which visualize SAT instances are examined for finding useful
possibilities to support the user's understanding. DPvis and SATIn were both developed
at University Tübingen and are presented in the next subsections.

1.1. DPvis

DPvis, which stands for Visualizing the Davis-Putnam Procedure, was developed by
Edda-Maria Dieringer and Carsten Sinz ([1]). The Davis-Putnam Procedure, also
refered to as DPLL-algorithm, is an algorithm which determines whether a
propositional logic formula is satisfiable. The purpose of DPvis is to analyse SAT
instances concerning their internal structure with regard to tractability. The run of a
DPLL-algorithm can be simulated. This is illustrated via two different views, namely
the variable interaction graph and a search tree. The user can step through each phase
of the DPLL-algorithm and is navigated by the search tree, which shows the state of
allocated variables. The variable interaction graph displays the remaining ones and
their relationships. If a valid assignment of values to variables, which satisfies the
formula, is found, this is shown to the user. As this visualization concept only
demonstrates the algorithm for testing the satisfiability of a formula and not the cause
of a possible failure, it creates no additional benefit for the investigated use case and no aspect is considered further.

1.2. SATIn

SATIn, SAT Insights abbreviated, is a tool for visualizing SAT instances and was developed by Stephan Kottler ([2]). The input, a SAT instance in cnf-format, is translated into different graphs, which can be chosen manually and are displayed contemporaneously. Variables, literals and clauses are set into relation by arbitrary combination. The interrelationships of the individual components become apparent. One possible user interaction is selecting one node in any graph, while the system responds by highlighting this specific node in any other graph displayed. Through these references, which are shown to the user, he is able to build a consistent mental model on the basis of different perspectives. This concept is adopted in the implemented prototype. Having these concepts as foundation, existing structures are combined in the developed prototype to form a complete visualization concept, which should support the user in understanding a proof, and thus the cause of the inconsistency in the development structures. Each of the used structures and their purposes are elucidated in detail in the next section.

2. Visualization structures

Different visualization structures are applied in the prototype to display various information. The individual structures vary in level of detail, interaction possibilities and kind of information. Adequate visualization concepts to increase understanding are used and described in detail in this section.

2.1. Feature Model

A basic graphical representation of development structures are feature models. They consist of a tree-like structure, where nodes represent the individual components, called features, and edges the interrelationships between those. Additionally there are cross-tree-constraints outside the edges, to express relationships between nodes which are not connected within the tree structure. The hierarchy determines the allocation between single features. The used notation and interpretation is adopted from Kang ([3]). Advantages of feature models are a simple, clear and packed visualization of all the components and their interrelationships. As the user should be confident with feature models, the visualization bases one these, hence the user is able to build comprehension upon a familiar structure. Another benefit of feature models is the possibility of translating them into formal language with the help of simple rules ([4]), thus automatic analyses are made easier. A disadvantage is that the cross-tree-constraints often cause inconsistencies, because they generate complexity as impacts are not directly obvious.
2.2. Proof

The proof which results from the performed analysis is based on resolution. As the feature model can be translated into formal language, namely into a set of given clauses, the proof consists of these clauses which resolve to several other clauses during resolution that finally can be resolved, if the model is invalid, to the empty clause. The proof is displayed as a set of clauses, not modified, shown in a simple listed text form.

Each line represents one clause with the following scope:

ID [REF_ID1, REF_ID2]: {LITERAL1, LITERAL2, ...}

The clauses are numbered consecutively with a unique identifier (ID), to allow referencing. The lines contain the literals of the clauses (LITERAL1, ...), and if the clause infers from two other clauses, the identifiers from the two latter are mentioned (REF_ID1, ...).

An example is given as follows:

Given a SAT instance

\[
\begin{align*}
\{ y, y, y, x, y, x \} & \\
& \rightarrow \{ y, x, y, \neg y, \neg y, y, \} \\
\end{align*}
\]

the corresponding proof is displayed in the following form:

\[
\begin{align*}
C_1 &: \{ x, y \} \\
C_2 &: \{ \neg x, y \} \\
C_3 &: \{ \neg y \} \\
C_4[C_1,C_2] &: \{ y \} \\
C_3[C_3,C_4] &: \{ \} \\
\end{align*}
\]

This visualization structure is used to give an overview of the complete proof. Experts, which have basic knowledge of formal languages and resolution based proofs, might just need this representation to understand the cause of the problem. For other user groups this information is processed, more specifically arranged in a reasonable order, and split into single steps. This is implemented with the following structures.

2.3. Proof Step

The proof step picks one specific clause and with the help of the referencing identifiers, shows these two resolving clauses. The content of all the clauses lets the user trace the resolution and does not overburden him with needless information. The understanding is supported by using the established notation of a step in a resolution-based proof. The notation can be seen for the resolution of clauses \( C_1 \) and \( C_2 \) from the example of the previous paragraph:

\[
\begin{align*}
C_1 &: \{ x, y \} \\
C_2 &: \{ \neg x, y \} \\
C_4 &: \{ y \} \\
\end{align*}
\]

With this small amount of information collected in a single view, the unexperienced user is not overstrained and is able to concentrate on one single step of the proof.
2.4. Resolution Graph

As resolution is the proof technique on which the analyses are built on, a visualization concept is needed. It is used to give an overview of the proof structure and its components. Thus this can be applied as navigation. The used resolution graph is a modification of the definition of Carsten Sinz ([5]), which is as follows:

Given a SAT instance \( S \) with the set of edges \( E \), undirected edges are drawn between clause \( C_1 \) and \( C_2 \), if one variable \( x \in E \) exists, so that \( x \in C_1 \) and \( \neg x \in C_2 \). Clauses which are adjacent in the resolution graph lead to a resolvent.

Example: Given a SAT instance \( S_1 = (C_1, C_2, C_3) = (\{x, y\}, \{\neg x, y\}, \{\neg y\}) \), the corresponding resolution graph is as in figure 1.

![Figure 1. Resolution Graph](image1)

This sort of resolution graph describes the connections of clauses in a model and reveals possible resolving variables. This graph only contains the existing clauses, which result from the model, without analysing it. As the resolution graph is used for manual navigation, the clauses which appear during the line of proof should also be visualized, as they are the ones of interest. This concept is achieved through a slightly modified resolution graph, which is comparable to the common notation of a resolution proof tree.

Given a SAT instance \( S_2 = (C_1, C_2, C_3, C_4, C_5) = (\{x, y\}, \{\neg x, y\}, \{\neg y\}, \{y\}, \{\}) \) the generated modified resolution graph is displayed in figure 2. If during the following text, resolution graph is mentioned, the one referred to is always the modified one.

![Figure 2. Modified Resolution Graph](image2)
3. Implemented Prototype

In this section the implemented prototype is presented. The scope of the graphical user interface is described, as well as application concepts, which are offered to the user. The employed framework for visualizing the graphs and other structures is mentioned.

3.1. Graphical User Interface

The graphical user interface is divided into four minor areas:

- Feature Model
- Resolution Graph
- Formal Proof
- Single Proof Step

A rough structure is recognizable in figure 3. The feature model takes half of the space as it represents the model and is a familiar structure for every user. The inconsistency results from an interrelationship within the model.

Figure 3. Structure of the graphical user interface

The individual concepts and interactions that help the user in understanding the proof, which are implemented in the prototype, are presented in the next section.

3.2. Application concepts

- Show/hide different views
  - Manually separate
  - Pre-built modes for certain user groups
- Collapse/expand parts of the feature model
- Zoom in/out of parts of the feature model
  - Continually via scrolling
  - Step by step via double click
- Shift of the feature model via drag’n’drop
- Interactive run-through of the proof and thus adaptation of the focus in all of the views
Sequentially go through the steps of the proof (backward and forward) by
- Clicking buttons of the graphical user interface
- Clicking left or right arrow key

Arbitrarily through selective choice of nodes in the resolution graph
- Highlighting of corresponding components in the individual views
- Fade-out/Hide irrelevant information like a subtree of the feature model

With these concepts the user is supported in understanding the line of proof. There are not only pre-built modes and predetermined visualization processes for unexperienced users, but users are also able to choose freely which views are shown and in which sequence the line of proof is visualized. The build process of a mental model can take place in different contexts concerning user experience, level of knowledge or information demands.

3.3. Framework

D3.js ([6]) is a framework based on JavaScript. It was developed mainly by Mike Bostock, Vadim Ogievetsky and Jeffrey Heer, derived from the framework Protovis. D3 stands for Data-Driven Documents which describes the application area. Its objectives are improving web based development with the main focus on dynamic and interactive data visualization. One advantage of D3.js is the direct access to DOM objects of the HTML document, hence direct manipulation of these is possible. Furthermore data can be bound to elements of the visualization, which initiates an automatic update, if data is modified. Automatic layout algorithms improve the intuitive understanding of the user.

D3.js fits to the considered use case, as it combines data with graphic visualization in a comfortable manner. Updates on the graphical user interface, are performed automatically, if data changes. D3.js offers layout algorithm for graph structure, as it is necessary for implementing this concept. The use of JavaScript as programming language and the web-based development creates a very flexible use context.

4. Further improvements

In this section a prospect on future increments is given, which implicate improvements or alternative implementations.

4.1. Proof Tree

As alternative to the proof, which is adopted unchanged from the input and listed in simple text form, an improvement to this representation is a structured, ordered presentation as a proof tree. The line of proof is combined with the former presentation. References to resolving clauses are used to position the clauses with content nearby. One node represents one clause. The child nodes are those who led to the parent clause during the resolution.

For an example of a proof tree see Figure 4.
The advantage is the structured representation. The user does not have to search for referenced clauses to understand a connection. A disadvantage might be displaying redundant information. One clause could be used for resolution of two other clauses, and using the described concept, it is therefore displayed twice.

![Proof Tree](image)

Figure 4. Proof Tree

Another benefit is the hierarchical structure. An introduced interaction might be collapsing or expanding parts of the proof. The user is able to hide information of no interest. Especially for large proofs this function is very useful, as parts of the proof can be hidden, and the user is able to concentrate on a small simple proof and join the results from those.

4.2. Constraint Interaction Graph

Some research was made about visualizing the structure of a SAT instance, but on a level which is more abstract as the variable interaction graph from DPvis, which was presented in section 1.1. Anthony Mak ([7]) introduces the Constraint Graph, which sets relation to variables and constraints. Some users of the developed tool might only be aware of cross-tree-constraints from the feature model, and if the line of proof is visualized with references to these structures, it might be a lot easier to understand for these unexperienced users. Four different constraint graphs are presented with the following exemplary set of clauses:

\[
\begin{align*}
C_1 : A & \Rightarrow B \\
C_2 : C & \Rightarrow (D \lor E) \\
C_3 : F & \Rightarrow C \\
C_4 : G & \Rightarrow A
\end{align*}
\]

Rossi ([8]) describes four different possibilities of relating clauses and variables graphically:

- Primal Constraint Graph or Variable View (see Figure 5)
  Edges indicate a common occurrence in one clause, which labels the edge.
• Dual Constraint Graph or Constraint View (see Figure 6)
  Edges indicate occurrence of one variable in several clauses.

• Constraint Hypergraph or Bipartite View (see Figure 7)
  Edges indicate the occurrence of a variable in a clause.

• Hyperplane View (see Figure 8)
  Circles incorporate occurring variables in the clause, which labels the circle.

These graphs could be integrated in the visualization concept, to build the understanding upon a level which is closer to the user. The logical interrelationships between components in the development structures become much clearer, thus the user is able to understand the problematic constraints and does not need to understand issues from another level of expertise.
4.3. Additional information

Another improvement might be displaying additional information, such as degree of edges, within the given graphs, but visualized with new concepts. Added value could be accomplished via ([7]):

- Size of nodes
- Colour of nodes
- Thickness of edges
- Colour of edges

5. Conclusion

A concept was developed, which supports different user groups in understanding the reason for inconsistent development structures. The visualization of the proof occurs in a user-oriented fashion, as unnecessary information is hidden and different interaction possibilities with the system are provided. These are depending on the user experience, level of knowledge and kind of information demand.

Further improvements are presented, which may be implemented in the future, to step to another level of meta language and offer more support to unexperienced users.

References