Tool Support for Incremental Consistency Checking on Variability Models

Michael Vierhauser[1] Deepak Dhungana[2] Wolfgang Heider[1] Rick Rabiser[1] Alexander Egyed[3]

[1] Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University
Linz, Austria
rabiser@ase.jku.at

[2] Lero - The Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
deepak.dhungana@lero.ie

[3] Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

Abstract—**The complexity of variability models makes it hard for product line engineers to maintain their consistency over time. Engineers need support to detect and resolve inconsistencies. In this paper, we describe our initial results towards tool support for incremental consistency checking on variability models. The main aim of our research is to improve the overall performance and scalability of consistency checking. We report on experiences of integrating an existing incremental consistency checker in the DOPLER product line tool suite.**

Keywords − variability models, incremental consistency checking, tool support.

I. INTRODUCTION AND MOTIVATION

Product line variability models are inherently complex. Independent of the modeling approach used (e.g., feature-oriented [1], decision-oriented [2], orthogonal [3]) real-world variability models can easily contain several thousand elements with diverse and often complex dependencies. Through the collaboration with our industry partner Siemens VAI − the world's leading company in engineering and plant-building for the iron, steel, and aluminum industries − we have learned that engineers in practice face big challenges in maintaining the consistency of variability models. The consistency of models is, however, essential for deriving correct products. It is also critical that variability models correctly reflect the actual system (e.g., components defined in the variability model must really exist). Therefore engineers should be supported in detecting and keeping track of inconsistencies during modeling.

Several consistency checking mechanisms have been reported in the literature and have been applied to various types of models [4], [5], [6]. A drawback of many of these approaches is that they are only capable of checking the consistency of entire models in a batch-oriented manner. This means that the consistency constraints are evaluated for the entire model at certain points in time (e.g., when saving a model). Due to the complexity of real-world models (often containing thousands of model elements and non-trivial dependencies), such a "batch-oriented" approach to consistency checking leads to performance problems. We also experimented with a batch-oriented approach in the context of our DOPLER product line engineering tool suite [7]. To improve performance, we however decided to incorporate an existing approach for incremental consistency checking of UML design models [8] in the context of product line variability modeling. In this paper, we describe our initial results towards tool support for incremental consistency checking on variability models.

II. CONSISTENCY CHECKING FOR DOPLER VARIABILITY MODELS

In collaboration with Siemens VAI we have been developing the decision-oriented product line engineering approach DOPLER [9].

A. *DOPLER modeling language*

DOPLER variability models comprise two elements: *Assets* and *Decisions*. Assets represent the core elements in the product line (e.g., components). Assets can depend on each other functionally (e.g., one component requires another component) or structurally (e.g., a component is part of a sub-system). DOPLER allows modeling assets at an arbitrary granularity and with arbitrary attributes and dependencies (based on a given set of basic types). Users can create domain-specific meta-models to define the types of assets, their attributes, and dependencies [9].

In case of Siemens VAI, the asset types that are part of variability models are components (representing Spring [10] XML component descriptions which in turn represent Java Beans), properties (key-value settings), resources (e.g., configuration files), as well as documents (e.g., user documentation). Diverse domain-specific dependencies have been defined, for example, a component can *require* other components, a component can *require* properties, or a document can *contribute to a* resource.

In DOPLER variation points are represented with *decisions*. Decisions have a unique name and a question that is asked to a user during product derivation. They can depend on each other hierarchically (if a decision needs to be taken before another decision becomes “visible”) or logically (if taking a decision changes the value of another decision). Possible types of decisions are Boolean, enumeration, string, and number.

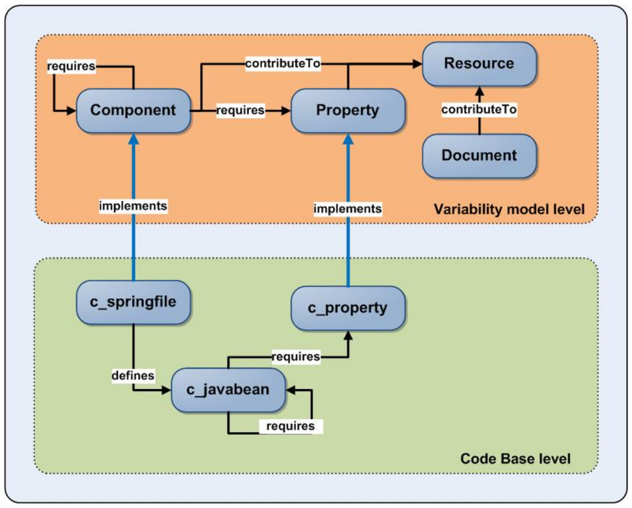


Figure 1. Siemens VAI meta-model overview. The upper part depicts the high-level meta-model. The lower part depicts additional elements needed to represent the code base of the product line. The relation of model level and file level is provided via the implements relation.

B. *Consistency Constraints*

When defining consistency constraints for Siemens VAI, our goal was to check consistency within the variability model as well as between the model and the code base of the product line. Our constraints therefore also check whether the model elements are consistent with concrete implementation artifacts like (Spring XML) component definitions and Java Beans. For constraints between the model level and the actual code base we generate a model representation of the code. For that reason the original DOPLER meta-model for Siemens VAI has been extended to cover information on the Spring files, the contained Java Beans, their properties, and the relations among the diverse elements (cf. Figure 1). Table 1 shows some examples of constraints that are relevant in the context of our industry partner.

We differentiate between generic and domain-specific (Siemens VAI) constraints. The generic constraints are relevant in any DOPLER variability model. For example, it is important to detect cycles between decisions (either based on hierarchal or logical dependencies among them) in any DOPLER model. These constraints are independent of the domain-specific meta-model (depicted in Figure 1). We therefore reuse these constraints and provide them as a core functionality of the consistency checker.

Siemens VAI-specific constraints mainly address model to code consistency. For instance, the most basic constraint SVAI1 assures that each component modeled in the variability model also exists in the code base of the product line. This constraint prevents, for example, that components that aren't available anymore or are outdated and therefore have been removed from the file system are not forgotten to be purged in the variability model as well. The two constraints SVAI2 and SVAI3 cover the relations between components in the model, and the relations between Spring XML files in the file system (these in fact depend on relations between the Java Beans described in that Spring XML files). Both constraints assure that there aren't any unnecessary relations between components respectively and that no relations are missing in the variability model.

TABLE I. EXAMPLES OF GENERIC (G) AND SIEMENS VAI-SPECIFIC (SVAI) CONSTRAINTS

	Constraints	
	Name	***Description***
G1	List decision	A list decision must have at least two options to choose from
G2	Mandatory attribute	Mandatory attributes must not be empty
G3	Decision effect cycle	There must be no cycles caused by logical decision dependencies
G4	Visibility condition cycle	There must be no cycles caused by hierarchical decision dependencies (visibility conditions)
G5	Visibility condition self reference	A visibility condition must not contain the decision itself
SVAI1	Component matching	Each component in the variability model must exist in the product line code base
SVAI2	Component relation	Relations between components in the variability model must also exist in the product line code base
SVAI3	Java Bean relation	A relation between Java Beans must be represented in the variability model as a component relation
SVAI4	Variant type relation	Variant types must not have requires relations
SVAI5	Variant type occurency	If two or more components are identical, all of them must contribute to a variant type component
SVAI6	Variant type consistency	Only identical components must contribute to a single variant type component

To illustrate how the defined constraints work we discuss constraint SVAI2 in detail by showing its high-level operation sequence: SVAI2 checks the necessity of *requires* relations between components. As illustrated in Figure 2, a *requires* relation between two components in the model is only needed if it is based on an existing dependency in the product line code base. Each component is "implemented" through a Spring file which in turn contains one or more Java Beans. If at least one Java Bean defined in Spring file 1 requires a Java Bean defined in Spring file 2, the relation on component-level is needed. Otherwise the consistency check will reveal the unneeded relation between component 1 and component 2.

This consistency constraint is not inherently complex to understand – indeed, most are of similar complexity. However, it is important to note that such consistency constraints may have to be evaluated many times in a model. For example, constraint SVAI2 needs to be evaluated for each requires relationship among two components and there are thousands of such requires relationships in our models.

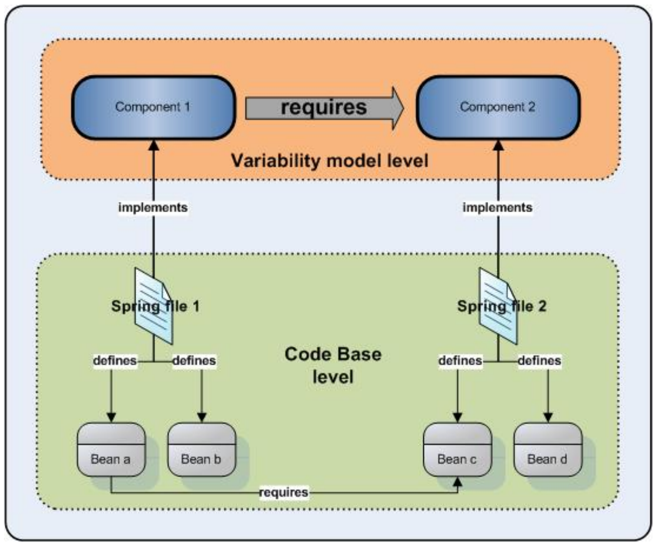


Figure 2. Schematic view of constraint SVAI 2

III. TOWARDS INCREMENTAL CONSISTENCY CHECKING SUPPORT FOR VARIABILITY MODELS

In our project with Siemens VAI, we developed a batch-oriented consistency checker early in the project. It worked fine as long as we were working with small variability models and a small number of constraints. The approach however didn't scale for very large models and a high number of required consistency checks. The performance problems did not allow to report inconsistencies to the user after each change to a model.

We therefore started exploring the use of an incremental consistency checker, which had been successfully evaluated for large UML models as part of the UML/Analyzer tool for instant consistency checking of UML models [8]. The tool helps designers in detecting and tracking inconsistencies correctly and quickly with every design change.

A consistency constraint needs to be re-evaluated if and only if one of the affected model elements changes. We refer to this set of model elements as the scope of a consistency constraint. Identifying the scope is simple in principle, however, it is not possible to predict in advance what model elements are accessed by any given consistency constraint.

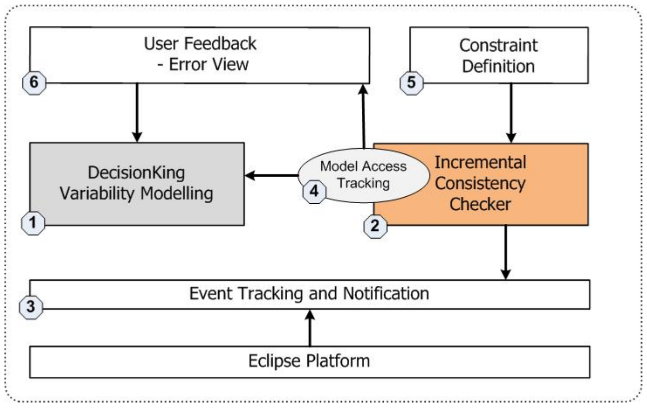
The UML/Analyzer tool circumvents this problem by observing the run-time behavior of consistency constraints during their evaluation. To this end, the equivalent of a profiler for consistency checking was developed. The profiling data is used to establish a correlation between model elements and consistency constraints (and inconsistencies). Based on this correlation, it then decides when to re-evaluate consistency constraints and when to display inconsistencies − allowing an engineer to quickly identify all inconsistencies that pertain to any part of the model of interest at any time.

IV. INTEGRATING THE INCREMENTAL CONSISTENCY CHECKER IN THE DOPLER TOOL SUITE

We have been integrating the incremental consistency checker approach in the Eclipse-based DOPLER tool suite. Figure 3 shows a high-level architecture of our tool and the main components of the checker.

The DOPLER variability model editor DecisionKing (#1) supports creating and updating variability models.

The *incremental consistency checker (#2)* performs constraint initialization, management, and persistence and applies incremental checking to variability models independent from the domain-specific meta-model used. This guarantees that the approach can later be easily used with other meta-models and is not limited to Siemens VAI.



User Feedback - Error View
6
Constraint Definition
5
DecisionKing Variability Modelling
1
Model Access Tracking
4
Incremental Consistency Checker
2
Event Tracking and Notification
3
Eclipse Platform

Figure 3. High-level tool architecture of incremental consistency checking within the DOPLER tool suite.

For instant consistency checking it is necessary to track user changes during modelling. An *event tracking and notification mechanism (#3)* allows observing changes to the variability model and the Eclipse workspace at a very detailed level. It provides information about DOPLER variability models being opened for editing and manages the propagation of change notifications from model elements to the incremental consistency checker [1]. As described in the previous section, incremental consistency checking highly depends on the ability of tracking and processing changes from various sources. The more fine-grained these change events can be tracked, the better performance can be achieved, because with each level of information detail fewer constraint instances eventually need to be evaluated. Our event tracking mechanism in the DOPLER tools allows to identify changes down to the level of model element attributes. Therefore, very few constraints need to be evaluated during incremental consistency checking.

The *model access tracker (#4)* monitors and logs all read access events to model elements for each single constraint instance. Each call on a model element by a constraint has to be done through a “model profiler”, which is capable of capturing and tracking any read access on attribute level. This evaluation profiling ensures that all necessary constraints are re-evaluated when a change event occurs.

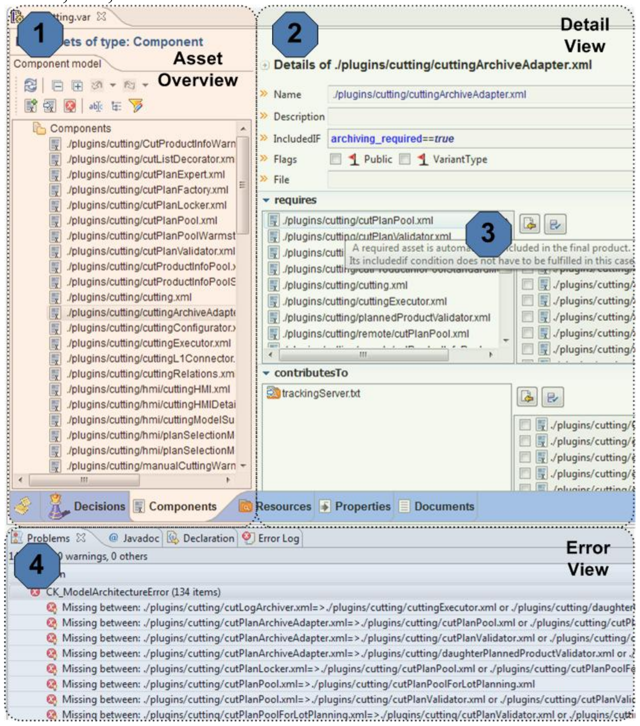
The consistency *constraint definition (#5)* uses the Eclipse extension point mechanism to add constraints to the incremental checking tool as different application domains need specific constraints. Note that our approach distinguishes the definition of a constraint from its evaluation.

The *error view (#6)* provides feedback to the users on the inconsistencies detected for the evaluated constraint. Eclipse provides the marker mechanism, which allows for easy creation and management of occurring errors. To assure a high level of flexibility, evaluation results are also provided through the extension point mechanism to make them utilizable in other plug-ins or in custom views.

V. APPLICATION EXAMPLE

A major goal when developing our incremental consistency checker was to achieve a better usability and responsiveness for the modeler working with the DOPLER tool suite and detecting and providing information on inconsistencies as early as possible. In the following we will describe a brief scenario of how a modeler can work with the tool, and in which way the tool supports detecting and fixing inconsistencies.

Figure 4 shows a screenshot of the DecisionKing variability model editor: In the Asset Overview (*#1*), the modeler can find an outline of already defined assets. Assets can also be added or removed here. The Detail View provides information about the currently selected asset, their attributes (*#2*), as well as relations to other assets (*#3*). An error view (*#4*) provides information on errors in the variability model, i.e., inconsistencies.



```
ets of type: Component
Component model
Components
./plugins/cutting/CutProductInfoWarn
./plugins/cutting/cutListDecorator.xm
./plugins/cutting/cutPlanExpert.xml
./plugins/cutting/cutPlanFactory.xml
./plugins/cutting/cutPlanLocker.xml
./plugins/cutting/cutPlanPool.xml
./plugins/cutting/cutPlanPoolWarmst
./plugins/cutting/cutPlanValidator.xml
./plugins/cutting/cutProductInfoPool.
./plugins/cutting/cutProductInfoPoolS
./plugins/cutting/cutting.xml
./plugins/cutting/cuttingArchiveAdapt
./plugins/cutting/cuttingConfigurator.
./plugins/cutting/cuttingExecutor.xml
./plugins/cutting/cuttingL1Connector.
./plugins/cutting/cuttingRelations.xm
./plugins/cutting/hmi/cuttingHMI.xml
./plugins/cutting/hmi/cuttingHMIDetai
./plugins/cutting/hmi/cuttingModelSu
./plugins/cutting/hmi/planSelectionM
./plugins/cutting/hmi/planSelectionM
./plugins/cutting/manualCuttingWarn

Details of ./plugins/cutting/cuttingArchiveAdapter.xml
Name         ./plugins/cutting/cuttingArchiveAdapter.xml
Description
IncludedIF   archiving_required==true
Flags        Public   VariantType
File
requires
./plugins/cutting/cutPlanPool.xml
./plugins/cutting/cutPlanValidator.xml
./plugins/cutti   A required asset is automa   cluded in the final product.
                  Its includedif condition does not have to be fulfilled in this case
./plugins/cutti
./plugins/cutting/cutting.xml
./plugins/cutting/cuttingExecutor.xml
./plugins/cutting/plannedProductValidator.xml
./plugins/cutting/remote/cutPlanPool.xml
contributesTo
trackingServer.txt

1 0 warnings, 0 others
CK_ModelArchitectureError (134 items)
Missing between: ./plugins/cutting/cutLogArchiver.xml=>./plugins/cutting/cuttingExecutor.xml or ./plugins/cutting/daughter
Missing between: ./plugins/cutting/cutPlanArchiveAdapter.xml=>./plugins/cutting/cutPlanPool.xml or ./plugins/cutting/cutP
Missing between: ./plugins/cutting/cutPlanArchiveAdapter.xml=>./plugins/cutting/cutPlanValidator.xml or ./plugins/cutting/c
Missing between: ./plugins/cutting/cutPlanArchiveAdapter.xml=>./plugins/cutting/daughterPlannedProductValidator.xml or ./
Missing between: ./plugins/cutting/cutPlanLocker.xml=>./plugins/cutting/cutPlanPool.xml or ./plugins/cutting/cutPlanPoolF
Missing between: ./plugins/cutting/cutPlanPool.xml=>./plugins/cutting/cutPlanPoolForLotPlanning.xml
Missing between: ./plugins/cutting/cutPlanPool.xml=>./plugins/cutting/cutPlanValidator.xml or ./plugins/cutting/cutPlanVali
Missing between: ./plugins/cutting/cutPlanPoolForLotPlanning.xml=>./plugins/cutting/cutPlanValidator.xml or ./plugins/cutti
```

Figure 4. DecisionKing Variability Modeling Tool and Error View displaying inconsistencies found after changes to the model.

A typical modeling process starts with defining new assets that are relevant in the newly created model. After defining all needed assets, the relations among them need to be modeled. The modeler can add or remove relations to other assets via drag & drop in the detail view *(#3)*. In contrast to the old batch-oriented approach, manipulating elements in the model now has an immediate effect. After adding a relation to an asset all involved constraints are being re-evaluated. Feedback about an inconsistency in the model is provided in that second the user takes the wrong action. The error view *(#4)* then provides detailed information on the occurring inconsistency and the source responsible for that. Assisted by this, the modeler can draw conclusions and resolve the occurring inconsistency by (in this example) removing an unneeded relation from the model.

VI. CONCLUSIONS

We presented initial tool support for applying incremental consistency checking on variability models. Our experiences with large-scale models demonstrate the performance and scalability of the approach. In future work we will increase the number of types of constraints and will also investigate how the dependencies among constraints can be exploited to further improve performance. Moreover, we will analyze the performance of the approach in detail.

ACKNOWLEDGMENTS

This work has been conducted in cooperation with Siemens VAI Metals Technologies and has been supported by the Christian Doppler Forschungsgesellschaft, Austria. This work has also been supported by FWF grant P21321-N15

REFERENCES

[1] K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," TR CMU/SEI-90TR-21, Carnegie Mellon Univ., Pittsburgh, PA, USA 1990.

[2] G. H. Campbell, Jr., S. R. Faulk, and D. M. Weiss, "Introduction To Synthesis," INTRO_SYNTHESIS_PROCESS-90019-N, Software Productivity Consortium, Herndon, VA, USA 1990.

[3] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, A. Vilbig: "A Meta-model for Representing Variability in Product Family Development". PFE 2003: 66-80.

[4] B. Belkhouche and C. Lemus, "Multiple View Analysis and Design, "Proc. of the Viewpoint 96: Int'l WS on *Multiple Perspectives in Software Development*, 1996.

[5] B. H. C. Cheng, E. Y. Wang, and R. H. Bourdeau, "A Graphical Environment for Formally Developing Object-Oriented Software, " Proc. of the *6th Int'l Conf. on Tools with Artificial Intelligence*, New Orleans, USA, 1994, pp. 26-32.

[6] A. Tsiolakis and H. Ehrig, "Consistency Analysis of UML Class and Sequence Diagrams Using Attributed Graph Grammars," Proc. of the *Graph Transformation & Graph Grammars*, Berlin, 2000, pp. 77-86.

[7] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer, "Integrated tool support for sw. product line engineering, "Proc. *22nd IEEE/ACM Int'l Conf. on Automated Sw Eng (ASE'07)*, pp. 533-534.

[8] A. Egyed, "Instant Consistency Checking for the UML," Proc. of the *28th Int'l Conf. on Sw. Eng. (ICSE)*, Shanghai, China, May 2006.

[9] D. Dhungana, P. Grünbacher, and R. Rabiser, "Domain-specific Adaptations of Product Line Variability Modeling, "Proc. of the *IFIP WG 8.1 Working Conf. on Situational Method Engineering*, Geneva, Springer Series in CS, 2007, pp. 238-251.

[10] R. Johnson, J. Höller, and A. Arendsen, "Professional Java Development with the Spring Framework," Wiley Publishing, 2005.

[11] W. Heider, R. Rabiser, D. Dhungana, P. Grünbacher, Tracking Evolution in Model-based Product Lines. 1st Int'l WS on *Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, Proc. (vol 2) of the 13th Int'l SW. Product Line Conf. (SPLC 2009), San Francisco, CA, 2009, Carnegie Mellon Univ., pp. 59-63.