Flexible and Scalable Consistency Checking on Product Line Variability Models

Michael Vierhauser
Christian Doppler Lab for Autom. Softw. Eng.
Linz, Austria
vierhauser@ase.jku.at

Paul Grünbacher
Systems Eng. and Automation
Johannes Kepler University
Linz, Austria
paul.gruenbacher@jku.at

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

Rick Rabiser
Christian Doppler Lab for Autom. Softw. Eng.
Linz, Austria
rabiser@ase.jku.at

Wolfgang Heider
Christian Doppler Lab for Autom. Softw. Eng.
Linz, Austria
heider@ase.jku.at

ABSTRACT

The complexity of product line variability models makes it hard to maintain their consistency over time regardless of the modeling approach used. Engineers thus need support for detecting and resolving inconsistencies. We describe a tool-supported approach for incremental consistency checking on variability models. Our approach significantly improves the overall performance and scalability compared to batch-oriented techniques and allows providing immediate feedback to modelers. It is flexible and extensible as new consistency constraints can easily be added. Furthermore, the approach is not limited to variability models and also checks the consistency of the models with the underlying code base of the product line. We present tool support and report the results of a thorough evaluation based on real-world product line models.

General Terms

Variability models, model consistency, incremental consistency checking.

1. INTRODUCTION

Product line variability models are inherently complex. Regardless of whether feature-oriented [1], decision-oriented [2], or orthogonal [3] variability models are used, their size represents a major challenge in real-world product lines as they can easily contain thousands of elements with diverse and often complex dependencies. In 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 learned that engineers in practice face big challenges when maintaining the consistency of variability models. The models are subject to continuous evolution [4] and need to co-evolve with the actual system they represent to maintain consistency. The consistency of the models is also essential for deriving correct products. In product line engineering consistency constraints range from simple rules (e.g., there must be no cycles in model element dependencies) and well-formedness criteria to more sophisticated checks (e.g., each component in the variability model must exist in the product line code base and vice versa). Engineers need support for detecting and keeping track of such inconsistencies when modifying the product line's models or code base.

Several consistency checking mechanisms have been reported in the literature and have been applied to various types of models [5–7]. However, there are several drawbacks with many of these approaches and tools. (i) They are typically only capable of checking the consistency of entire models in a batch-oriented manner meaning that the relevant consistency constraints can be evaluated at certain points in time only (e.g., when saving a model) due to the complexity of the models. For example, when using a batch-oriented approach in our DOPLER product line tools [8] we ran into significant performance problems when working with real-world variability models of our industry partner which contain thousands of model elements with non-trivial dependencies and mappings. The bad performance meant that feedback could not be provided to modelers immediately as launching the batch checker after each change to the model turned out to be infeasible. Checking consistency after each save of the model has another disadvantage. Depending on how many changes engineers made since the last check many inconsistencies are displayed at once making it much harder for modelers to relate their modeling actions with the reported inconsistencies. (ii) Another drawback of many available consistency checkers is their lack of extensibility. It is complicated for modelers to add new consistency constraints and to remove or modify existing ones. We also faced this challenge with our batch-oriented checker as the various consistency concerns were hard to identify and separate in the code. (iii) Finally, in product line engineering it is essential to ensure consistency between models and code. Changes to the product line are made either to the variability models or the actual code base. In both cases developers

or modelers can introduce inconsistencies that need to be detected. Existing checkers are often limited to either models or to code and do not check across boundaries.

To address these challenges, we started exploring the use of incremental consistency checking for product line variability modeling. Our research goals were to improve performance to allow immediate feedback to modelers; to improve the extensibility by making it easy to add, remove, or modify consistency constraints; and to extend the scope of consistency checking to model-to-code consistency.

While we benefitted from our earlier experiences with an existing incremental checker for UML design models [9] we faced several research challenges when adapting it to model-based product lines. For instance, checking consistency with code required a common reasoning basis for model and code. We thus created a model as an image of the code and applied incremental transformation to keep the image synchronized with the code thereby allowing the consistency checker to operate on the model image instead of the code itself.

The tool features we developed as part of this research are briefly summarized in a short workshop paper [10]. Here, we describe details about the approach and its implementation and report results of a thorough evaluation. We first introduce different types of possible inconsistencies on and among different layers of product line models. To illustrate the practical challenges, we describe consistency constraints we developed for the component-based product line of our industry partner. We then present our solution for incremental consistency checking on variability models and describe its integration in an existing product line engineering tool suite. We evaluate the approach using product line models of our industry partner and provide an evaluation with regard to performance and scalability. We discuss related work and conclude the paper with an outlook on future work.

2. CONSISTENCY CHECKING IN MODEL-BASED PRODUCT LINES

Product line models cover both the problem space and the solution space [11] as shown in Figure 1. A problem space model describes the variability of a system using concepts from the problem domain, e.g., in the language of the users in a specific domain. A solution space model provides a representation of the variable system from a technical point of view, e.g., it describes optional or alternative artifacts and their dependencies [12].

Consistency needs to be maintained between these modeling spaces but also within each space. However, in order to be useful, product line models also need to be consistent with the actual code base. Checking and maintaining consistency with the code base is particularly challenging as it largely depends on the implementation technology used. Furthermore, the code base is in practice often changed by developers outside the product line modeling environment.

Existing variability modeling approaches cover the upper two levels in Figure 1. Feature models, decision models, or orthogonal models have been proposed for modeling the problem space [1–3]. Architecture description languages or general purpose languages like the UML can be used to define the solution space. There are also approaches such as DOPLER [13] or CML (Concept Modeling Language) [12] that provide modeling support for both levels and mappings

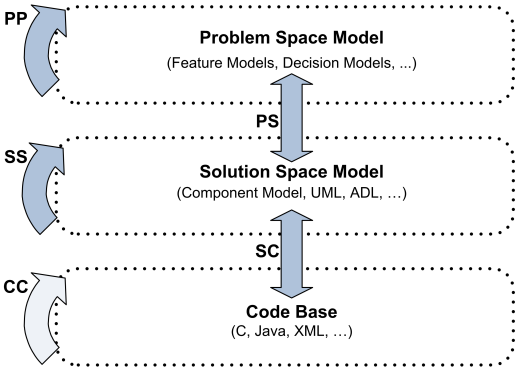

Figure 1: Modeling levels in product lines with types of intra- and inter-level consistency checks.

between them. Figure 1 shows the three modeling spaces in product line engineering together with five different types of consistency checks:

PP inconsistencies can exist within the problem space model. For instance, in a decision model cycles caused by decision dependencies need to be detected and prevented.

PS inconsistencies can occur between the problem space model and the solution space model. For instance, a component in the solution space model can have dependencies to two contradicting decisions. This can result in an inconsistency if it is impossible to resolve both decisions during product derivation.

SS inconsistencies can exist within the solution space model such as contradicting dependencies between components. One component might require two other components which in turn exclude each other.

SC inconsistencies can appear between the solution space model and the code base it represents. Such inconsistencies can for example occur whenever an asset (or its dependencies or attributes) are changed in the solution space model but not in the code base and vice versa.

CC inconsistencies can exist within the code base. Such inconsistencies are however typically addressed by integrated development environments and are thus not further investigated in this paper.

2.1 Modeling a Component-based Product Line with DOPLER

In cooperation with our industry partner Siemens VAI, we have been creating variability models for the CC-L2 software product line controlling continuous casting machines in steel plants. The (simplified) meta-model for modeling the problem space, the solution space and the code base is shown in Figure 2. The product line has been modeled using the DOPLER approach which uses decision models to define the problem space, asset models defining reusable elements of different types and their dependencies for the solution space, as well as mappings among decisions and assets [13, 14].

Variation points are represented using *Decisions* which have a unique name and a question that is asked to a user during product derivation. Answering a question sets the value of a decision. Possible answers depend on the type of the decision (Boolean, enumeration, string, or number). The range of allowed values can be further restricted by validity

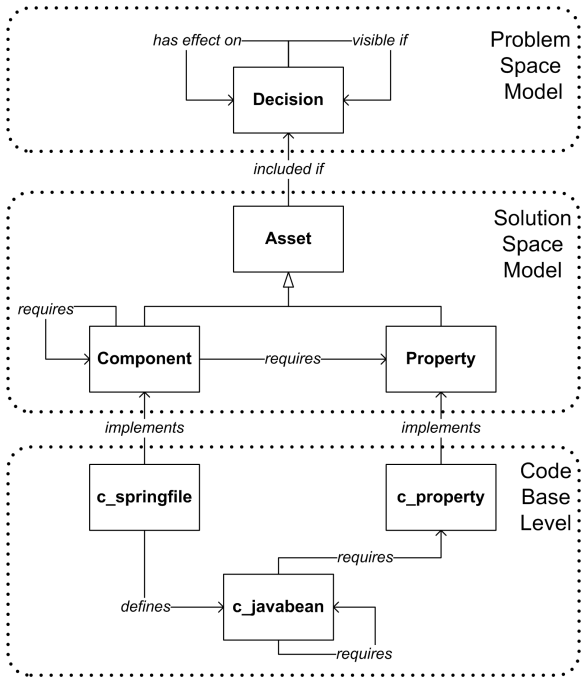

Figure 2: Partial Siemens VAI meta model with model elements for the problem space and solution space. The code base level defines additional elements for representing the component-based implementation of the product line.

conditions. Decisions 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).

Assets represent the solution space in the product line (e.g., software components). Assets can depend on each other functionally (e.g., one component requires or excludes another component) or structurally (e.g., a component is part of a sub-system). DOPLER allows modeling assets at arbitrary granularity and with user-defined 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. In the Siemens VAI product line the asset types in the DOPLER variability models are components representing Spring XML component descriptions which in turn represent Java Beans and properties (key-value pairs). Diverse domain-specific dependencies have been defined. For example, a component can require another component or a property (cf. Figure 2). Dependencies between assets and decisions are explicitly modeled via inclusion conditions that define for an asset when it will be part of a derived product. Due to asset dependencies, not every asset needs to be related with a decision to be included in a derived product. For example, when an asset is included because of a decision, its *requires* dependencies might lead to the inclusion of other assets as well.

We decided to create a model image of the code base and apply incremental transformation to synchronize the model image with the code. This allows us to use the same checking mechanism for all different types of artifacts. For that reason the meta-model was extended with a *code base level* containing the Spring files *(c_springfile)*, the contained Java Beans *(c_javabean)* and their properties *(c_property)*, as well as *defines* and *requires* relations among these elements (cf. Figure 2). Traceability to the solution space model is achieved by the *implements* relationship.

2.2 Examples of Consistency Constraints

Figure 1 shows generic types of constraints (PP, PS, SS, SC, CC) in product line engineering. Here we describe a number of specific constraints based on these types needed in the component-based product line of our industry partner. Examples of the constraints are shown in Table 1. To allow arbitrary checks and freedom for the developer, the constraints are defined in the Java programming language. There are, however, no language restrictions and any constraint language could be used instead.

The problem space and problem to solution space constraints (PP and PS) are relevant in any DOPLER variability model. There are simple constraints that check whether decisions of type enumeration have at least two possible values (PP1) and that values for model elements with mandatory attributes have been defined (PP2). It is also essential to detect cycles in the decision model stemming from hierarchical and logical dependencies (PP3–5). PS1 is an example of a problem-to-solution space inconsistency. The constraint detects whether inclusion conditions exist in the solution space (assets) that never can evaluate to true because they refer to contradictory decisions in the problem space.

Further constraints depend on the domain-specific meta-model for Siemens VAI (cf. Figure 2) and address intra-solution-space and model-to-code consistency. SS1 checks that each component modeled in the solution space (representing a Java Bean described in a Spring XML file) requires at least one property defining initialization parameters for that component. The most basic model to code constraint SC1 assures that each component modeled in the variability model exists in the code base of the product line. This constraint assures for example, that outdated components that are no longer available in the workspace are marked to be purged in the variability model as well. The constraints SC2 and SC3 cover the relations between components in the model, and the relations between Spring XML files in the file system (which in fact depend on relations between the Java Beans described in that Spring files). Both constraints assure that there are no unnecessary relations between components and that no relations are missing in the variability model. Constraints SC4-6 assure the consistency of variant type components which represent a particular characteristic of the Siemens VAI models. Variant types are used to group identical components implemented in different parts of the code base (different Spring files).

We discuss constraint SC2 in detail to illustrate its high-level operation sequence: SC2 checks the *requires* relations between components. As illustrated in Figure 3 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 defined by a spring file which in turn is realized by one or more Java Beans. If at least one Java Bean contained in the Spring file *(Deburrer.xml)* requires a Bean defined in the second Spring file *(Caster.xml)* then the relation is needed on component level. Otherwise

Table 1: Examples of constraints in component-based product lines.

Constraint	*Description*
PP1: Enumeration decision	An enumeration decision must have at least two options to choose from.
PP2: Mandatory attribute	Mandatory attributes of model elements must not be empty.
PP3: Decision effect cycle	There must be no cycles caused by logical decision dependencies.
PP4: Visibility condition cycle	There must be no cycles caused by hierarchical decision dependencies (visibility).
PP5: Visibility condition self reference	A Visibility Condition must not contain a reference to itself.
PS1: Inclusion condition exclusion	An Inclusion Condition must not contain two contradictory decisions (to avoid assets that will never be included in a derived product).
SS1: Component properties	Each component (representing a Java Bean) requires at least one property defining initialization parameters for the Bean.
SC1: Component matching	Each component in the variability model must exist in the product line code base.
SC2: Component relation	Relations between components in the variability model must also exist in the product line code base.
SC3: Java Bean relation	A relation between Java Beans must be represented in the variability model as a component relation.
SC4: Variant type relation	Variant types must not have *requires* relations.
SC5: Variant type occurrence	If two or more components are identical all of them must contribute to the same variant type component.
SC6: Variant type consistency	Only identical components must contribute to a single variant type component.

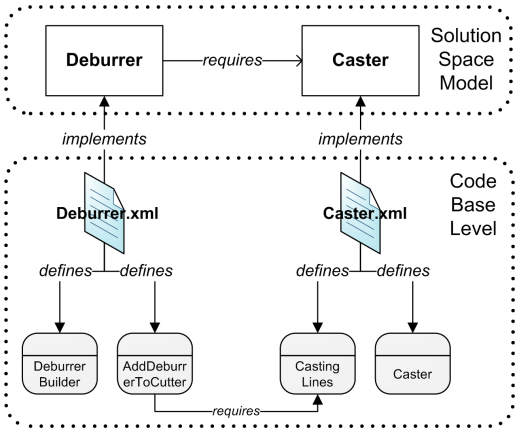

Figure 3: Schematic view of Constraint SC2.

the consistency check will reveal an obsolete relation between the two components. Although SC2 is fairly simple it is important to note that such constraints have to be evaluated many times in complex models. For example, SC2 needs to be evaluated for each *requires* relationship among two components (and there are thousands of such relationships in our models).

3. INCREMENTAL CONSISTENCY CHECKING ON VARIABILITY MODELS

Earlier in our research we had developed a batch-oriented checker that worked well with small variability models and a limited number of constraints. However, this approach did not scale well enough for larger models and high number of required consistency constraints that we encountered in industrial practice. In particular, modelers requested immediate feedback regarding consistency after changes to the model. While technically it would have been possible to realize immediate feedback with a batch checker, the computational cost made it impractical to report inconsistencies after each change. Previously, our approach thus only reported inconsistencies after a user invoked the checker when saving the model in the tool. Typically many changes were made by modelers between two invocations of the checker and multiple new errors were reported at once. This made it difficult for modelers to relate the errors to the changes they made to the model or in the code base – changes that were made minutes or even hours earlier. Also, our batch-oriented checker was difficult to extend or adapt because the consistency constraints were woven together into a large, complex algorithm. This was a big obstacle to the practical use of consistency checking because new constraints were increasingly hard to integrate in the existing algorithm.

We thus integrated an incremental consistency checker [9] to help modelers with detecting and tracking inconsistencies correctly and quickly after every change. The incremental checker is fully automated and does not require manual as-

sistance. It can be used to provide consistency feedback in an intrusive or non-intrusive manner. However, thus far the checker had only been used and evaluated for UML models that seem to be similar to our models, however exhibit quite a range of differences which required significant extensions. For example, as discussed in Section 2 and shown in Table 1 many of our consistency constraints go beyond the checking of models and also include source code and other artifacts. Since the existing approach was limited to checking models, we had to augment it with incremental transformation to automatically and quickly transform code fragments into model fragments to enable their checking.

To support the fast incremental checking of model changes, our checker [9] identifies all model elements that affect the truth value of any given consistency constraint (consistency is a Boolean state). A consistency constraint needs to be re-evaluated if one of these model elements changes. We refer to this set of affected model elements as the scope of a consistency constraint. The elements contained in the scope are stored in the scope database. Identifying the scope is simple in principle, however, it is not possible to predict it in advance.

Our incremental consistency checker computes the change impact scope of a constraint instance automatically by observing the run-time behavior of consistency constraints during their evaluation. To this end, we developed the equivalent of a profiler for consistency checking. The profiling data is used to establish a correlation between model elements and consistency constraints. Based on this correlation, it then decides when to re-evaluate consistency constraints and when to display inconsistencies – thus allowing an engineer to quickly identify all inconsistencies that pertain to any part of the model of interest at any time.

We illustrate the incremental consistency checker using consistency constraint SC1 (cf. Table 1) evaluating whether a modeled component is implemented in the source code. Figure 4 depicts a (simplified) excerpt of one of our models with three components in the top and a range of code-level constructs at the bottom. Consistency constraint SC1 is written from the perspective of a component – i.e., a given component is considered consistent if it is implemented by a code-level XML file. Since there are three components in Figure 4, each component has to be evaluated with regard to SC1. Our approach thus instantiates the consistency constraint SC1 three times, once for each component (ci1, ci2, ci3). We do this because incremental consistency checking needs to react to model changes and each instance of the consistency constraint is affected differently by model changes. This means that we need to compute how each consistency constraint or instance thereof is affected by a model change. For this purpose, we need to know the complete change impact scope for every constraint instance. Figure 4 indicates that the three SC1 instances ci1, ci2, and ci3 access distinct model elements when they are evaluated.

For example, the first instance *ci1* starts its evaluation at component *Deburrer*, then navigates along the *implements* relationship, and finally accesses *Deburrer.xml*. The constraint instance is satisfied because the component does have an associated code-level XML file. Our approach recognizes that from the entire model only these accessed model elements were needed to evaluate instance *ci1*. This instance must be re-evaluated only if one of those elements changes. Our approach thus maintains a table of instances of consistency constraints and the model elements they accessed during their evaluation to understand which instances must be re-evaluated when the model changes. There are a range of subtle issues (e.g., a chicken and the egg problem in that there must be an initial evaluation to build up the table before the incremental mechanism can take over). These issues are discussed in [9] and are out of the scope of this paper. It is also important to note that a model element may be accessed by multiple instances of one or more consistency constraints. For example, constraint SC2 evaluates a component with regard to the *requires* relationship. It makes sure that a component-level *requires* relationship is matched by a code-level *requires* relationship. Figure 4 depicts a *requires* relationship from *Deburrer* to *Caster* and during evaluation an instance of SC2 will access the components, their code-level XML files *Deburrer.xml* and *Caster.xml* and some of the *defines* and *requires* relationships underneath. The change impact scope of this instance of SC2 is also automatically observable through the model profiler. This scope is also larger than the scope of SC2 and encompasses much of the model elements underneath *Deburrer* and *Caster* but not *NozzleCheck*. Some model elements, such as *Deburrer.xml* thus affect multiple instances of consistency constraints. Only these related instances must be re-evaluated if *Deburrer.xml* changes.

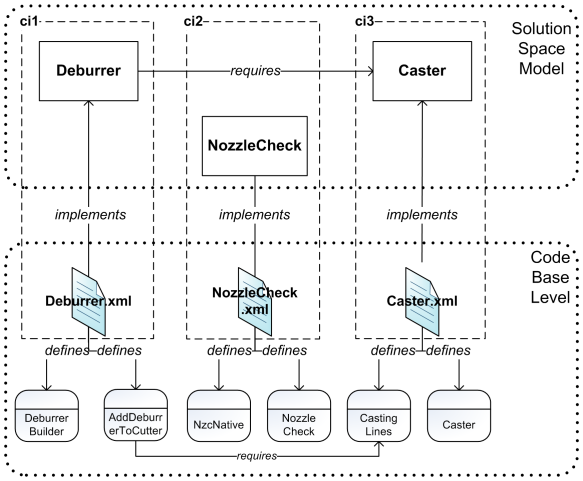

Figure 4: Incremental Evaluation of SC1 and SC2.

4. USING THE INCREMENTAL CHECKER IN THE DOPLER TOOL SUITE

A major goal when developing our incremental consistency checker was to increase usability by providing immediate feedback to modelers about the detected inconsistencies. We thus decided to seamlessly integrate the incremental consistency checker in the Eclipse-based DOPLER tool suite [8] which supports product line variability modeling and product derivation. In particular, we integrated the checker with DOPLER's modeling tool that allows defining decision and asset models as discussed in Section 2.

A modeler can define assets in the variability model editor as shown in Figure 5. The Asset Overview shows an outline of already available software components. This view allows adding or removing components. Selected components can

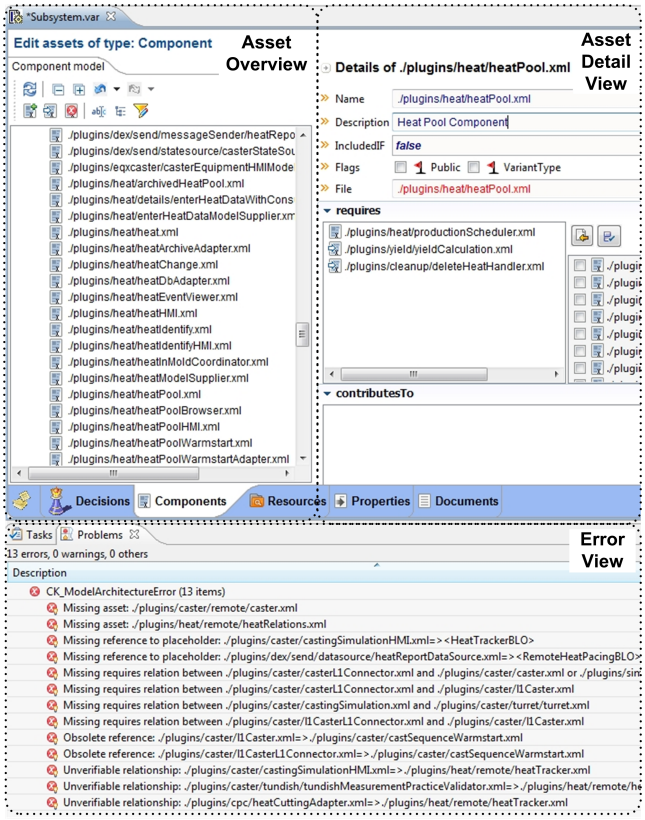

Figure 5: The DOPLER Variability Modeling Tool. The Incremental Consistency Checker identifies errors and displays them in an Error View.

be modified in the Asset Detail View which provides information about its attributes and relations to other components. The modeler can simply add or remove relations to other components via drag and drop in the detail view. Manipulating components in the editor has an immediate effect. For instance, after adding a relation to another component all involved constraints (and only those!) are re-evaluated. Feedback about detected errors is provided within milliseconds after the user action leading to an inconsistency. The Error View provides information about all inconsistencies found in the model. The Error View also provides details about the source model elements causing the problem. This helps the modeler to resolve the problem, for instance, by removing an unneeded relation between two components from the model.

The existing incremental consistency checker was originally developed for the Rational Rose UML tool suite, while in this project it had to be integrated in the DOPLER tool suite. Numerous adaptations were required to embed the checker which cannot be described here due to space constraints.

The DOPLER Variability Model Editor allows creating and updating variability models as described above. We illustrate our tool architecture by discussing the chain of events and data flow (cf. Figure 6).

The incremental consistency checker relies on tracking changes in the Eclipse workspace. For that purpose, our

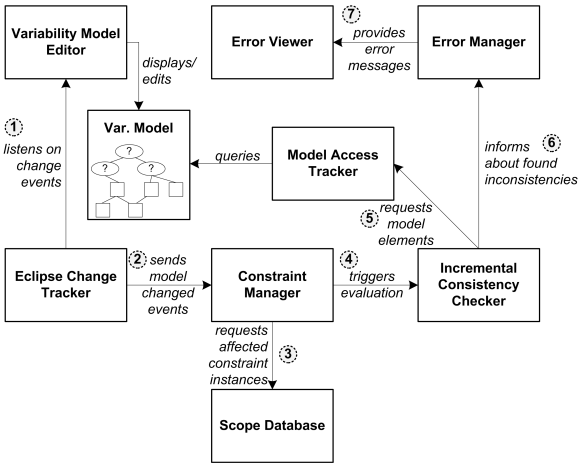

Figure 6: The Incremental Consistency Checker re-evaluates constraint instances after changes to the model.

Eclipse Change Tracker [15] observes arbitrary changes to the variability model and the Eclipse workspace. For instance, it listens to the "delete component" change to the variability model after the modeler deletes a component in the editor and propagates the change to the Constraint Manager which is responsible for initializing, managing, and storing constraints. Using the Scope Database the Constraint Manager determines the constraint instances to be re-evaluated after the deletion of a certain component. Our architecture makes use of the Eclipse extension point mechanism to allow adding new constraints flexibly and easy for new domains. Constraint definitions can easily be added or removed from the evaluation process by simply activating or deactivating plugins.

The list of constraint instances is used by the Incremental Consistency Checker which applies incremental checking to variability models by controlling the execution of all affected constraint instances. The constraint instances do not directly access and query the Variability Model. Instead, they use the Model Access Tracker – a model adapter component which monitors and records all read access events to model elements for each single constraint instance that is evaluated. This fine-grained "model profiling" helps improving performance, because with each level of detail fewer constraint instances need to be evaluated eventually. The Model Access Tracker also builds the Scope Database ensuring that only necessary constraints are re-evaluated after changes to the model (not shown in Figure 6).

The incremental consistency checker sends all newly found inconsistencies to the Error Manager which manages a list of errors for the user. It translates abstract constraint evaluation results into error messages meaningful for a modeler and performs basic filtering functions to avoid information overload. Finally, the Error Viewer (see also Figure 5) displays the detected inconsistencies for the evaluated constraint. We use the Eclipse marker mechanism for the easy management of errors and their presentation in a viewer.

5. EVALUATION

We define our evaluation goal using the Goal-Question-

Metric (GQM) approach [16]. A goal in GQM consists of a purpose describing the actual aim of the evaluation, the issue of interest, a specific object to be measured and the viewpoint which describes the stakeholder perspective from which the goal is defined. Goals are refined using questions targeting explicit problems of interest. Finally, metrics are defined to obtain data addressing those questions. *Our evaluation goal is to assess (purpose) the performance (issue) of incremental consistency checking (object) from the viewpoint of the modeler (viewpoint).* We explore three questions regarding performance, memory usage, and scalability of the approach.

Question 1.1 *"What impact does the incremental consistency checking technique have on the start-up performance?"* is further refined using two metrics: M 1.1.1 is the time needed to initialize the incremental consistency checker (measured in milliseconds). It measures the overall start-up time when opening a variability model and includes library initializations as well as the creation of the scope database. The startup time is critical as engineers frequently open models in the DOPLER tools. M 1.1.2 measures the initial memory footprint of the incremental consistency checker after its start-up expressed by the number of objects created (i.e., constraint instances, scope elements). Understanding memory consumption is important as developers modeling real-world product lines have to work with very large models containing thousands of model elements.

Question 1.2 *"What are the specific performance characteristics of the incremental consistency checker during modeling?"* is further defined with metric M 1.2.1 measuring the time needed (in milliseconds) for consistency checking of key atomic modeling tasks such as adding, modifying, or deleting model elements. This metric is important as it shows the feasibility of the approach during everyday modeling activities. In particular, it shows whether the approach can provide "immediate feedback" to modelers.

Question 1.3 *"How well does the approach scale?"* investigates performance and memory consumption of our approach when applied to industrial models of different sizes. We compute the metrics M 1.1.1., M 1.1.2, and M 1.2.1 for real-world models of different sizes to address this question. In addition we use another metric to investigate this question. M 1.2.2 measures the number of constraint instances that need to be evaluated for the atomic modeling actions. This metric checks whether the approach scales regardless of the specific modeling actions performed by the user.

5.1 Evaluation Setup

We used variability models of the Siemens VAI CC-L2 product line to perform our evaluation. Regarding question 1.3 we evaluate the incremental checker with different model sizes. We thus created models of different sizes by incrementally merging the available variability models using the merging approach described in [4]. The smallest variability model (I) represents a single subsystem and consists of 57 components. We use two other models II and III for evaluation with 200 and 401 components. The largest model IV consists of 768 components. The underlying code base used for evaluation contains 1956 code model elements (702 XML spring files and 1254 Java Beans) and is used in all four models I-IV.

Table 2 provides a detailed overview of the four models regarding the number of components (# C's), number of instantiated constraints (# CI's), and the total number of instantiated scope elements (# SE's) at startup. A constraint is instantiated for each model element only if needed as explained in Section 3. The scope database contains all scope elements defining the constraint instances that need to be re-evaluated after a change. The actual number of model elements is much higher in these models as we only show assets of type of Component in the table and do not consider the number of relations and model element attributes in Table 2.

Table 2: Siemens VAI solution space models used for evaluation.

	Included subsystems	*# C's*	*# CI's*	*# SE's*
I	Cutting	57	1044	2084
II	Model I subsystems + Caster, Heating	200	1902	3385
III	Model II subsystems + Optimizer, JAMP, Simulator	401	3108	5180
IV	Model III subsystems + Analysis, DefectTracking, VAIQfeeder, Warmstart	768	5310	8175

5.2 Evaluation Results

We present the results gained by comparing the four different models in terms of startup and runtime performance when applying the incremental consistency checker.

The mechanism for incremental consistency checking relies on a complete execution of all constraints during startup to initialize the constraint instances and the scope database (scope elements need to be created). Figure 7 provides an overview of the initialization costs of the incremental checker for the four different models (lower solid line). The additional time needed to open a variability model lies between 2,5 seconds for a model with about 200 assets, and 5 seconds for a model with 400 assets. This time is clearly acceptable for typical model sizes at our industry partner (100 to 400 components per single model). Also, this task is performed as a background thread when opening a model and does not block the modeler (even in the case of model IV where the initialization takes about 14 seconds).

The lower solid line is essential for the evaluation of the incremental checker. However, we made an additional interesting observation. Re-structuring the constraints already implemented in our legacy batch checker into the new format required by the incremental checker already had a significant impact on performance. We compared the complete time of executing all constraint instances with the initialization time of our legacy batch-checking approach. The dashed line demonstrates the scalability issues we had experienced with the batch-checking approach. Please note, however, that the legacy checker is not fully comparable in terms of the constraints it checks.

Besides the initialization time, we also took a closer look at the memory costs (M 1.1.2). Memory consumption depends on the number and type of constraints and the number of model elements. The maximum number of constraint instances is created if every constraint is instantiated once for every asset in the model and every asset in the code base (this is only a theoretical case). Instead, constraints are typically instantiated for selected types of asset only (e.g.,

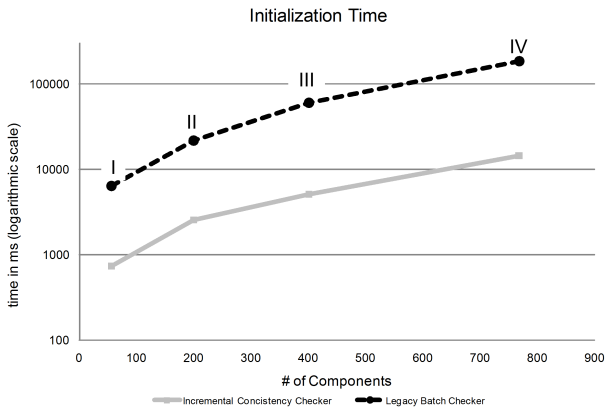

Figure 7: Metric M 1.1.1. Initialization time of the new incremental checker and evaluation time of legacy batch checker for models I-IV.

components) and the real number of constraint instances per model is therefore much lower than the theoretical maximum.

The maximum number of scope elements can be calculated as the number of assets times the number of attributes per asset. Scope elements are created only once, but can be used in multiple constraint instances. Figure 8 shows the linear increase of scope elements and constraint instances for the four variability models demonstrating the scalability of our approach.

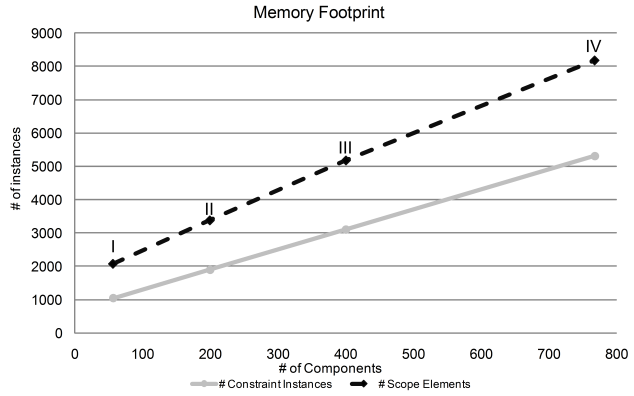

Figure 8: Metric M1.1.2. Number of constraint instances and scope elements after startup (models I-IV).

In Section 4.1 we described a number of basic tasks when working with the DOPLER variability modeling tool. We use these typical actions on variability models to gain data concerning the time needed to evaluate model changes at runtime. We simulate creating and deleting assets, as well as adding and removing relations for our live evaluation (metric M 1.2.1) and the analysis of affected constraints during one evaluation cycle (metric M 1.2.2).

For live evaluation analysis each single task has been performed 100 times to outweigh effects such as unpredictable

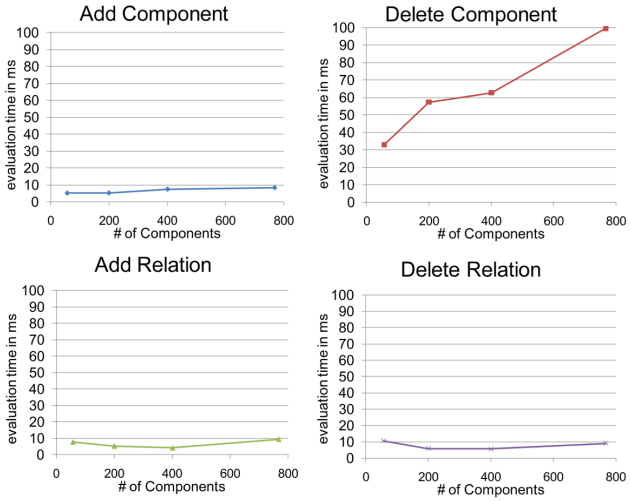

Figure 9: Metric M 1.2.1. Performance of basic modeling actions for models I-IV.

background tasks. The mean value gained is shown in Figure 9. The costs for adding assets, as well as adding and removing relations to/from an asset remains constant in terms of model size. A single action performed on a variability model took only about *5 to 10 ms* which demonstrates that our approach indeed provides *immediate feedback*.

Only the task of deleting an asset from a model increases linearly. This is because constraints that were instantiated for the asset to be deleted, need to be found and removed from the consistency checking instance. We expect however that this problem can be easily fixed by using a different data structure.

While constraint instances and scope elements in total increase depending on model size (as shown in Figure 8), the number of constraint instances that need to be re-evaluated during live evaluation stays constant. Depending on the type of change very few constraints need to be evaluated (see Figure 10). The number of constraints that need to be evaluated remains constant when adding assets in models of different size. Adding and deleting relations as well as deleting component moves between 2.5 and 3.2 constraints evaluated per change event, again tested on 100 change samples.

6. RELATED WORK

Several papers address the issue of consistency between models and code in product lines. Approaches for product line evolution try to avoid the erosion of a product line, i.e., the deviation from product line models up to the point where key properties no longer hold, e.g., [17]. Murta *et al.* [18] present an approach for ensuring consistency of architectural models to implementation during evolution. Their approach support arbitrary evolution policies and is based on recording changes in a configuration management system. Product line evolution support becomes critical in model-based approaches to ensure consistency after changes to meta-models, models, and actual development artifacts. Mende *et al.* [19] describe tool support for the evolution of software product lines based on the "grow-and-prune"

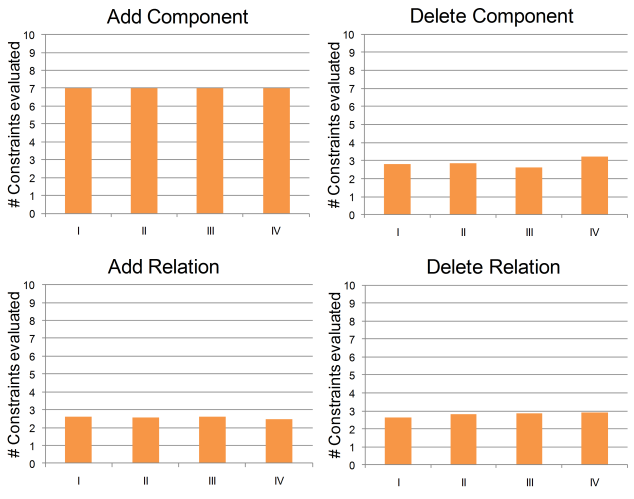

Figure 10: Metric M 1.2.2. Number of constraint instances affected by a change for models I-IV.

model. They support identifying and refactoring code that has been created by copy & paste and which might be moved from product to product line level. Deng *et al.* [20] describe a model-driven product line approach that addresses the issue of domain evolution in product line architectures with model transformations.

While researchers generally agree on the importance of consistency checking, the methods on how to detect detect them vary widely. For example, Tsiolakis and Ehrig [7] present an approach for checking the consistency between class and sequence diagrams based on a common graph structure. Van der Straeten *et al.* [21] use description logic to detect inconsistencies between sequence and state chart diagrams. Campbell *et al.* [22] use a model checker to evaluate inconsistencies within and across UML diagrams. Zisman and Kozlenkov [23] use a knowledge base and express consistency rules using patterns and axioms.

Several approaches are based on identifying inconsistencies between different design models by direct comparison. Some of these approaches also perform incremental consistency checking. xLinkIt [24] allows evaluating the consistency of XML-based documents. The approach can check the consistency of entire UML models but can also handle incremental consistency checking by only evaluating changes to versions of a document. ArgoUML [25] detects inconsistencies in UML models based on annotated consistency rules that also enable incremental consistency checking. The approach implements two consistency checking mechanisms: consistency rules without annotations are placed into the queue which is continuously evaluated in the background using a batch-checker at 20% CPU time. Consistency rules with annotations are evaluated using an incremental checker. It has been demonstrated that ArgoUML's type-based consistency checking produces good performance but it is not able to keep up with an engineer's rate of model changes in very large models [26]. Similar to our approach Blanc *et al.* [27] address the issue of incremental consistency checking from the perspective of model changes. However, their consistency rules need to be defined explicitly in terms of their impact on changes. If done correctly this leads to good performance. However, since writing these annotations may easily cause errors, they are no longer able to guarantee the correctness of incremental consistency checking.

While it is important to know about inconsistencies, it is often too distracting to resolve them right away. The notion of "living with inconsistencies" [28] advocates that there is a benefit in allowing inconsistencies in design models on a temporary basis. While our approach provides inconsistencies instantly, it does not require the engineer to fix them instantly. Our approach tracks all presently-known inconsistencies and lets the engineer explore inconsistencies according to his/her interests in the model.

7. CONCLUSIONS AND FUTURE WORK

We presented a tool-supported approach and evaluation results of applying an incremental consistency checker on product line variability models. The incremental consistency checker works on and across different levels of variability models and also checks consistency between variability models and source code. The incremental consistency checker is independent from the domain-specific DOPLER meta-model and can be easily used with arbitrary meta-models. As our event tracking mechanism allows to identify changes down to the level of model element attributes only few constraints need to be evaluated during incremental consistency checking. Our evaluation with large-scale models demonstrates the performance and scalability of the approach. It is fast enough to provide immediate feedback to users and identifies errors within 5 to 10 ms for typical modeling actions. Furthermore, our approach allows adding new constraints in a flexible manner via Eclipse extension points. Constraints can be activated and deactivated as needed in certain domains.

In future work we will experiment with more and other types of constraints. We will also investigate how the dependencies among constraints can be exploited during constraint evaluation. Finally, we plan to extend our tools to support fixing identified inconsistencies as already demonstrated in [29].

Acknowledgements

This work has been supported by Siemens VAI Metals Technologies, the Christian Doppler Forschungsgesellschaft, and the Austrian FWF grant P21321-N15. We are also grateful to Martin Lehofer and Deepak Dhungana for their support.

References

[1] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., TR-21, Software Engineering Inst., CMU, Pittsburgh, PA, USA, 1990.

[2] G. H. Campbell, S. R. Faulk, and D. M. Weiss. Introduction to Synthesis. Tech. rep., Software Productivity Consortium, Herndon, VA, USA, 1990.

[3] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In F. van der Linden (Ed.), *5th Int'l WS on Sw. Product-Family Engineering*, vol. LNCS 3014, pp. 66–80. Springer, Siena, Italy, 2003.

[4] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software (to appear)*, 2010.

[5] B. Belkhouche and C. L. Olalde. Multiple view analysis of designs. In *Proc. of the 2nd Int'l Software Architecture WS (ISAW-2) and Int'l WS on Multiple Perspectives in Software Development on SIGSOFT '96 workshops*, pp. 159–161. ACM, New York, NY, USA, 1996.

[6] B. H. C. Cheng, E. Y. Wang, and R. H. Bourdeau. A graphical environment for formally developing object-oriented software. In *ICTAI*, pp. 26–32. 1994.

[7] A. Tsiolakis and H. Ehrig. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In *Proc. of the Graph Transformation and Graph Grammars (GRATA), Berlin, Germany*, pp. 77–86. 2000.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE'07)*, pp. 533–534. ACM, Atlanta, Georgia, USA, 2007.

[9] A. Egyed. Instant consistency checking for the UML. In *28th Int'l Conf. on Software Engineering*, pp. 381–390. ACM, New York, NY, 2006.

[10] M. Vierhauser, D. Dhungana, W. Heider, R. Rabiser, and A. Egyed. Tool support for incremental consistency checking on variability models. In *Proc. 4th Int'l WS on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, pp. 171–174. ICB-Research Report 37, Univ. of Duisburg Essen, Linz, Austria, 2010.

[11] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[12] K. Czarnecki. Variability modeling: State of the art and future directions (keynote presentation). In *Proc. 4th Int'l WS on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, p. 11. ICB-Research Report 37, Univ. of Duisburg Essen, Linz, Austria, 2010.

[13] D. Dhungana, P. Heymans, and R. Rabiser. A formal semantics for decision-oriented variability modeling with DOPLER. In *Proc. 4th Int'l WS on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, pp. 29–35. ICB-Research Report 37, Univ. of Duisburg Essen, Linz, Austria, 2010.

[14] P. Grünbacher, R. Rabiser, D. Dhungana, and M. Lehofer. Model-based customization and deployment of eclipse-based tools: Industrial experiences. In *24th IEEE/ACM Int'l Conf. on Automated Sw. Eng., Auckland, New Zealand*, pp. 247–256. 2009.

[15] W. Heider, R. Rabiser, D. Dhungana, and P. Grünbacher. Tracking evolution in model-based product lines. In *1st Int'l Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2009), Proc. (vol 2) of the 13th Int'l Software Product Line Conference (SPLC 2009)*, pp. 59–63. SEI CMU, San Francisco, CA, USA, 2009.

[16] V. Basili, G. Caldiera, and D. Rombach. Goal/question/metric paradigm. In J. Marciniak (Ed.), *Encyclopedia of Software Engineering*, pp. 528–532. John Wiley and Sons, New York, 1994.

[17] S. Johnson and Bosch. Quantifying software product line ageing. In P. Knauber and G. Succi (Eds.), *Proc. of the 1st ICSE 2000 WS on Software Product Lines: Economics, Architectures, and Implications*, pp. 27–32. Limerick, Ireland, 1994.

[18] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner. ArchTrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In *21st IEEE/ACM Int'l Conf. on Automated Software Engineering, Tokyo, Japan*, pp. 135–144. 2006.

[19] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *12th European Conf. on Sw. Maintenance and Reengineering*, pp. 163–172. IEEE CS, Washington, DC, USA, 2008.

[20] G. Deng, J. Gray, D. Schmidt, Y. Lin, A. Gokhale, and G. Lenz. Evolution in model-driven software product-line architectures. In P. Tiako (Ed.), *Designing software-intensive systems*, pp. 1280–1312. Idea Group Inc (IGI), 2008.

[21] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proc. 6th Int'l UML Conference, San Francisco, CA, USA*, pp. 326–340. 2003.

[22] L. A. Campbell, B. H. C. Cheng, W. E. McUmber, and K. Stirewalt. Automatically detecting and visualising errors in UML diagrams. *Requir. Eng.*, 7(4):264–287, 2002.

[23] A. Zisman and A. Kozlenkov. Knowledge base approach to consistency management of UML specification. In *16th IEEE Int'l Conf. on Automated Software Eng., San Diego, CA, USA*, pp. 359–363. 2001.

[24] 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.

[25] J. Robins et al. ArgoUml, http://argouml.tigris.org/. Tech. rep.

[26] A. Egyed and D. S. Wile. Support for managing design-time decisions. *IEEE TSE*, 32(5):299–314, 2006.

[27] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *30th Int'l Conf. on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pp. 511–520. 2008.

[28] R. Balzer. Tolerating inconsistency. In *Int'l Conf. on Software Engineering (ICSE)*, pp. 158–165. 1991.

[29] A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Proc. of the 2008 23rd IEEE/ACM Int'l Conf. on Automated Sw. Eng.*, pp. 99–108. IEEE Computer Society, Washington, DC, USA, 2008.