# Fixing Configuration Inconsistencies Across File Type Boundaries

Christoph Elsner
Siemens Corporate Technology
Erlangen, Germany
christoph.elsner.ext@siemens.com

Daniel Lohmann, Wolfgang Schröder-Preikschat
Friedrich-Alexander University
Erlangen-Nuremberg, Germany
{lohmann,wosch}@cs.fau.de

***Abstract*—Creating a valid software configuration often involves multiple configuration file types, such as feature models, domain-specific languages, or C header files with preprocessor defines. Enforcing constraints across file types boundaries already at configuration is necessary to prevent inconsistencies, which otherwise are costly to discover and resolve later on.**

**We present a pragmatic framework to specify and apply inconsistency-resolving fixes on configuration files of arbitrary types. The framework converts each configuration file to a model, checks it for consistency, applies fixes, and serializes it back again. We argue that conventionally programmed fixes and round-trip mechanisms (i.e., converters and serializers) are indispensable for practical applicability and can provide sufficient reliability when following usual development practices. We have developed round-trip mechanisms for seven different configuration file types and two fixing mechanisms. One fixing mechanism extends previous work by combining automatic detection of correct fix locations with a marker mechanism that reduces the number of locations.**

**A tool-supported process for applying the fixes provides user guidance and integrates additional semantic validity checks on serialized configuration files of complex types (e.g., feature models). Evaluations reveal a speed up in inconsistency fixing and that the performance of the currently integrated round-tripping and fixing mechanisms is competitive.**

## I. Introduction and Motivation

Creating a consistent configuration (e.g., of a software product line) often affects various types of configuration files with subtle dependencies. Whereas customer-visible variability might be bound via selecting feature model options, the deployment of software to physical nodes may reside in domain-specific models or text files,while fine-tuning is done via preprocessor variables in C header files. Choosing certain feature model options may constrain choices in the domain-specific model, while setting a preprocessor variable in turn may presuppose a feature to be set. Configuration complexity increases further when employing configurable off-the-shelf components (e.g., Apache, Oracle) or when building combined products including other product lines, which in turn expose their variability via certain configuration file formats.

In previous work, we have developed an infrastructure [1] that enables converting arbitrary configuration files to models in order to define and check cross-configuration-file constraints using model-based constraint languages (e.g., as OCL expressions). Yet, our approach has required fixing each reported inconsistency manually in the original configuration file. We improve on that in this paper by contributing the design of an extensible framework for (semi-)automatically applying inconsistency fixes directly on model level and serialize the changes back to the original configuration files.

The framework converts each configuration file to a model, checks it for consistency, applies fixes, and serializes it back again. Basically, this paper gives answers to three questions: (1) How to develop reliable round-trip mechanisms (i.e., model converters and serializers) both for simple (e.g., Java property files) and more complex (e.g., feature models) configuration file formats? (2) Which fix mechanisms can we provide for simple (one element to change) and complex fixes (dozens of elements to change)? (3) How can we support the user during fix application in the best possible manner when dealing with complex fixes and complex file formats?

After outlining the framework in Section II, we give answers to this questions:

(1) In Section III, we will show that existing tooling and common development practices enabled us to develop reliable round-trip mechanisms for various configuration file formats (Ecore DSMs, XText DSLs, XMLSchema XML, Java property files and C header files with #defines) with reasonable effort. For some complex file formats (e.g., pure::variants[1] feature models) we can, however, not avoid that semantic inconsistencies are introduced (e.g., that constraints defined in the feature model are violated) when applying fixes on the model.

(2) In Section IV, we argue that, although imperatively programmed fixes are unavoidable for complex fixes, many simpler fixes can correctly be derived from the constraint that checks for an inconsistency. Therefore we integrate both an imperative mechanism and an adapted version of a derivative fixing mechanism into the framework.

(3) In Section V, we present a tool-supported process for applying the fixes. It supports the user when dealing with imperatively programmed fixes and complex file formats by providing user control and guidance functionality and by incorporating additionally semantic validity checks on the serialized configuration files.

We implemented the framework, which currently incor-

[1] http://www.pure-systems.com/, visited 2011-05-22.

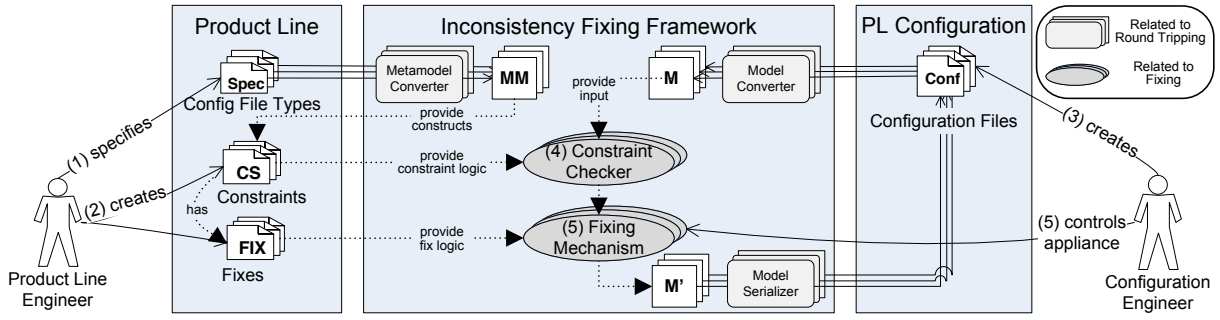


Fig. 1. Product line engineer and configuration engineer using the inconsistency fixing framework.

porates seven round-trip and two fixing mechanisms, as an Eclipse extension. We performed evaluations indicating that its performance is competitive and that the fix application process speeds up inconsistency fixing compared to manual adaptation of configuration files (Section VI). Finally, we discuss and conclude the paper (Sections VII and VIII).

## II. Outline of the Framework

In this section, we give a general outline of the configuration inconsistency fixing framework. It builds on the general assumption that each artifact involved in product line configuration can be mapped into the modeling world and back again (round tripping). Hence, modeling serves as a pivot technology for defining constraint checks across file type boundaries and for applying fixes in case of constraint inconsistencies.

In the following, we will first introduce general modeling terminology and its correspondence to product line configuration. Then, we outline the framework, which reveals that the round-trip and fixing mechanisms constitute its crucial parts.

### A. Modeling Terminology

A *model* is a formal abstraction of a concept (e.g., a physical system or software) describing its concrete entities and relationships [2]. The formal rules, which specify the entity and relationship *types* allowed in a certain model, are provided by its *metamodel*. As an example, a simple metamodel for modeling #defines in a C header file will comprise the root entity type `DefineList`, which contains an arbitrary number of `Define` elements, which in turn have a `name` and a `value` element of type string. A model conforming to this metamodel, for instance, defines a concrete DefineList comprising one Define with `name = "REDUNDANT_FC"` and `value = "1"`. For specifying metamodels, a *metamodeling technology* is used (e.g., Eclipse Ecore [3]).

Constraints and fixes on models usually are specified using the elements defined in the metamodel. An expressive metamodel providing meaningful construct names and types is therefore essential for easy definition of constraints and fixes.

### B. Product Line – Model Correspondence

For simplification, we merely regard a software product line as a piece of configurable software. The configuration of a concrete product of the product line is described by its current *set of configuration files* (e.g., web-server configuration files, C header files, or domain-specific models and text files). The product line itself defines, either implicitly or explicitly, the *set of configuration file types* it can deal with (the web-server configuration language, a domain-specific grammar, etc.).

For the purpose of constraint checking and fixing, it is necessary to map each artifact involved in product line configuration to its corresponding representation in the modeling world. As we also argued in [1], each configuration file can be mapped to a *model*; the elements and the relations allowed in the certain configuration file (i.e., the *configuration file type*, such as the configuration file grammar) are converted to the *metamodel* of this model. Summarizing, it is necessary to transform the concrete *configuration* of a product line to a *set of models*, the *product line* itself to a *set of metamodels*.

### C. Usage Setup for the Framework

Given the previous assumptions, Figure 1 outlines the usage setup of our inconsistency fixing framework for principally arbitrary configuration file types. It involves two roles: *product line engineer* and *configuration engineer*. The former specifies the set of possible configuration file types a product line can deal with (1). Our framework converts them to metamodels using *metamodel converters*. Then, the product line engineer implements the domain constraints and fixes (2), thereby using the constructs and types defined in the metamodels. The configuration engineer, in turn, creates the configuration files of a concrete product (3), which our framework transforms to models conforming to the metamodels using *model converters*.

The framework then constraint checks the generated models (4) in the background. For each constraint that does not hold, the configuration engineer may choose to apply fixes (5) associated with the constraint. Finally, a *model serializer* component writes the changes back to the original file.

The development of reliable round-trip mechanisms (i.e. the (meta-)model converters and serializers) and practical checking and fixing mechanisms constitute crucial parts of the framework. We will discuss these issues in Section III and IV, respectively.

## III. Developing Round Trip Mechanisms

In this section, we will first argue why metamodel converters are a crucial feature for practical applicability of our fixing

approach. We then show how to pragmatically develop and make use of tools in order to engineer round-trip mechanisms of reasonable quality for seven different file types. Afterwards, we address related work and discuss which future research is needed in order to integrate two further promising approaches of recent research: lenses [4] and M3-level bridges [5].

### A. Metamodels for Efficient Fixing

As mentioned in Section II-A, constraints on a model are defined using the constructs defined on metamodel level. With a suitable metamodel, constraints become more concise and editors can provide richer support. For example, for feature models, we create a dedicated metamodel element for each feature, together with properly-typed attributes. Thus, we can leverage the type checking and tab completion of the constraint editor—typing errors in features and attributes become immediately evident. As we will see in the following section, existing tools also follow this paradigm of having dedicated metamodel converters. This helps us to create round-trip mechanisms for various configuration file types by reusing mature tools.

### B. Round Tripping for Various File Types

We developed metamodel converters and model converters and serializers for seven configuration file types (cf. Table I). In the following, we address their implementation.

| | Source files for MMs / Ms | Converters and Serializers MM Conv. | M Conv./Ser. |
|---|---|---|---|
| **Specific file types** | | | |
| Ecore Models | *.ecore/*.xmi | Java (EMF) | Java (EMF) |
| XMLSchema | *.xsd/*.xml | Java (EMF) | Java (EMF) |
| XText Grammars | *.xtext/*.mydsl | Java (EMF) | Java (EMF) |
| P::V Feature Models | *.xfm/*.vdm | Java (XML) | Java (XML) |
| KConfig Language | KConfig.*/*.config | plain Java | plain Java |
| **Generic file types** | | | |
| CPP Header | (none)/*.h | (fixed MM) | Java (+CPP) |
| Java Property Files | (none)/*.prop | (fixed MM) | plain Java |

KConfig: Configuration language used, e.g., for Linux kernel configuration
MM: Metamodel | M: Model | CPP: C preprocessor
EMF: Eclipse Modeling Framework API | P::V: pure::variants

TABLE I
CONVERSION AND SERIALIZATION STRATEGIES APPLIED TO SEVEN CONFIGURATION FILE TYPES.

**Source Files.** We distinguish generic and specific file types (as also done in [1]). For specific file types, a product line engineer can provide a specification file (e.g., a pure::variants feature model (*.xfm), an XMLSchema definition (*.xsd) or an XText[2] grammar specification (*.xtext)), which a metamodel converter will map to a *specific metamodel*. For other file types, which we call "generic", a product line does not formally specify which constructs (e.g., Java properties, CPP defines) are considered as valid. For those file types, we use a *generic metamodel*, which is identical for all product lines. When deriving a product, the configuration engineer creates concrete configuration files, (a header file (*.h), a feature model configuration (*.vdm), or an XML file (*.xml)). Our inconsistency fixing framework maps each configuration file transparently (as an IDE background process) to a *model*.

[2]http://www.eclipse.org/Xtext/, visited 2011-05-22.

**(Meta-)model Converters and model Serializers.** The table shows how we developed (meta-)model converters and model serializers for various specific and generic file types.

*Specific File Types.* The converters that were easiest to develop were those for Ecore domain-specific models, XMLSchema-based XML files, and domain-specific languages adhering to XText textual grammars. For each of the file types, we leverage the EMF framework, which provides a metamodel converter, which maps the specification of the configuration file type (in *.ecore, *.xsd, or *.xtext format) to a corresponding metamodel. EMF also provides an API for converting and serializing the corresponding configuration files (*.xmi, *.xml, *.mydsl).

For feature models in pure::variants format, we drive a pragmatic conversion approach. The feature model defines all allowed constructs (features and attributes). The developed metamodel converter simply maps each feature to a model element type in the metamodel (cf. Figure 2, right). All relationships between features (excludes, implies, etc.) are ignored. We also developed appropriate model converters and serializers that map the feature model configuration (an "instance" of the feature model) to a model and vice versa. The converter for KConfig works very similar: Each configuration item in a KConfig specification file is mapped to a model element type in the metamodel, each chosen configuration option in the *.config configuration file is mapped to the corresponding model element.

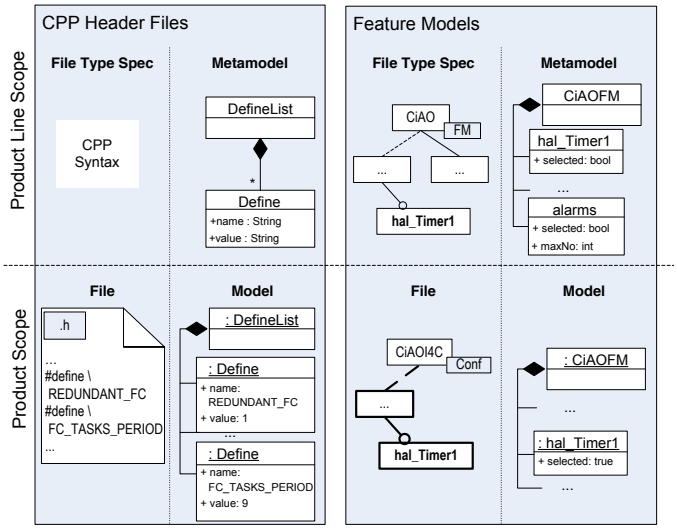


Fig. 2. Mapping file types and files to metamodels and models for CPP files and feature models.

As the file formats of pure::variants and KConfig are rather complex, we implemented the converters and serializers in plain Java. Although we can say, due to testing, that the serializers create syntactically correct configuration files (*.vdm, *.config), we cannot ensure that the configuration is *semantically valid* in the sense that all constraints defined in the respective specification files (such as, "FeatureA excludes FeatureB") are still met. As these constraints can become

very complex (pure::variants, for example, allows defining constraints between features as arbitrary Prolog statements), we currently refrain completely from trying to ensure them on model level. This, however, means that these constraints remain unnoticed unit serializing back to the original file type. To detect and recover from such semantic inconsistencies, our framework will check for semantical validity only *after* the fixed models have been serialized back to their original file format, thereby delegating the "tricky" part to the respective original tool. More details will be provided in Section V.

*Generic File Types.* In contrast to the aforementioned file types, CPP header files and Java property files are *generic* file types. This is, each model representing the #defines in a header file will have the *same* metamodel, as there is no specification file determining all valid #define symbols including their value types. The metamodel basically only defines a list of defines with names and values of type string (cf. Figure 2, left). To avoid complexity, we impose stringent restrictions on CPP header files when used for configuration via #defines. In particular, we do not allow multiple definitions of the same symbol or #defines within #ifdef, such as "#ifdef A .. #define B". Existing code needs to be refactored appropriately. For Java property files, which basically are lists of name-value pairs, too, the metamodel has the same structure as for CPP header files.[3]

### C. Discussion of Round Trip Research

There are many research approaches for converting files of principally arbitrary type into a processable format ("models" in the wider sense), transforming them, and serializing them back again. In particular we will consider those approaches that provide bidirectional "views" on the original data, as well as so called M3-level bridges. We argue that, while both approaches provide techniques that may become useful in future versions of our framework, no "killer feature" is missing that hampers its current practicability.

Bidirectional approaches [6], such as the lenses approach [4] or triple graph grammars (TGGs) [7], allow deriving both, converters and serializers, from a complex to a more *simple* "model" and vice versa using only one single expression (a lens or a mapping graph, respectively). This eliminates asymmetries between converters and serializers due to improper implementation. However, a lens (or a mapping graph) is tightly coupled with the metamodel it can map onto. While they might be used to implement model converters for *generic* configuration file types, where the metamodel is fixed, they cannot be used to deal with specific configuration file types (e.g., feature models), where we require metamodel converters which create a *different* metamodel for each distinct feature model. Although our current generic file types, CPP files and Java property files, are sufficiently simple so that using such techniques was not necessary, we consider integrating them in later versions of the framework, as they might support more complex generic file types.

[3]Content that cannot be parsed as a name-value pair is saved as plain text, so that no information gets lost during round tripping.

On the other hand, there are approaches for explicitly establishing an automated bridge between two "metamodels of different metamodeling technologies", so called M3-level bridges (e.g., [5]). For instance, when we already consider the original feature model to be a "metamodel", which just happens to be defined within another metamodeling technology (the feature model language), what we need to build are M3-level bridges from the feature modeling language to the modeling technology of our fixing framework (Eclipse Ecore) and vice versa. However, the current M3 approach is not yet formalized and rather defines a methodology with mapping guidelines. Therefore, it cannot guarantee symmetric model converters and serializers, as lenses or TGGs can. Furthermore, the M3 approach aims to match the concepts of two modeling technologies as far as possible. As shown in [8], a one-to-one mapping between feature modeling and metamodeling can only be done by artificially introducing new, semantically unclear constructs into feature modeling (e.g., feature inheritance). What might be needed is an adaptation of the symmetric M3-level approach that also supports semantic simplification, for example, as done in Figure 2 (right) were we intentionally refrain from mapping the constraints defined in the feature model to the metamodel.

To sum up: Both approaches provide useful techniques, view-based approaches for developing generic converters and serializers guaranteed to be symmetric, M3-level bridges as a method to engineer bridges between semantically similar metamodeling technologies. Although both are highly relevant and can become useful in future versions of our framework, their current lack does not hamper its practicability.

## IV. Model Fixing Mechanism

In this section, we will present the fixing capabilities of our framework. As model fixes repair inconsistencies, which are detected by constraint checks, we first illustrate the requirements for fixing by presenting some example constraints. We argue that an imperative mechanism for specifying fixes is crucial to deal with arbitrary complex fixes. Then, we discuss the state of the art in model checking and fixing, from which we adapt a further fixing mechanism [9] for automated derivation of correct fix locations from a constraint. This makes it possible to define simpler fixes declaratively instead of imperatively.

### A. Example Constraint

Figure 3 shows three exemplary constraints in the Xpand Check language [10], which is similar to OCL. The metamodels and models are created from the configuration files of our quadrotor helicopter evaluation platform [1] transparently to the user in an IDE background process. For this purpose, the converters for C headers (`hwHeader` and `swHeader`), feature models (`ciaoVDM`), and XMLSchema (`ciaoXML`) presented in Section III-B have been used.

All three constraints concern flight control redundancy. It is activated when setting the #define `REDUNDANT_FC` in a header file. The first constraint checks whether the #define `FC_TASKS_PERIOD` (the total period of all flight control

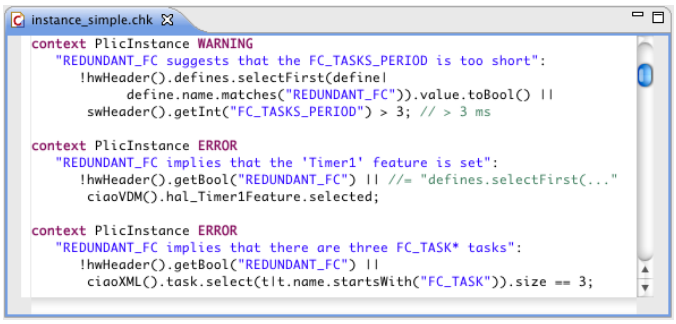


Fig. 3. Three constraints regarding flight control redundancy (`REDUNDANT_FC`).

tasks) is plausible in case `REDUNDANT_FC` is selected. The second constraint ensures that the timer support is activated in the feature model configuration. Finally, constraint three checks whether three redundant flight control XML tasks structures have been configured (plausibility check on task names).

The characteristics or the example are, to our experience, quite representative. Many configuration option, such as those in constraint one and two, work on simple data types (boolean, integer). Fixing corresponding inconsistencies basically means switching a boolean or integer value. Some fixes, however, can become very complex. In order to fix inconsistency three, it is necessary to create three task structures and set various additional task properties. In particular to support such complex fixes, we see the need to support arbitrary complex, imperatively-defined fixes on model level.

### B. Imperative Fixing Mechanism

For imperative fixes, we developed a mechanism to annotate each Xpand Check constraint with an arbitrary number or `SUGGEST` clauses. The `SUGGEST` annotation is parameterized with a textual description and an Xtend [10] model-transformation function:[4]

```
SUGGEST(String descr, XtendFunction action)
```

The implementation is based on a separate preprocessor step. The preprocessor records the `SUGGEST` clauses (the fixes) attached to each constraint and removes them before executing the constraint via the unchanged constraint checking engine.

We implemented the constraint checking and fixing mechanism to work incrementally using the technique presented in [9]. All constraints are initially checked once and all model elements each constraint accesses are recorded. When a constraint evaluates to false, it is marked as inconsistent. Then, any later time, the user may open a Fixing GUI in the IDE. For each inconsistent constraint, the Fixing GUI presents the textual `descriptions` of its attached `SUGGEST` clauses. On selection, the desired fix (the Xtend `action` function) is applied. By also recording which elements the fix changes, we only need to incrementally reevalute those constraint that had touched one of these elements in their last run.

[4]Xpand Check actually constitutes a subset of the Xtend language.

### C. Inconsistency Fixing: Related Work

Several different approaches combine constraint checking with fixing facilities, which can guarantee some desirable properties. Xiong et al. [11], for example, designed the constraint language Beanbag, which describes both constraints and fixes in one single expression. Whereas they can guarantee correctness of the fixes (i.e., the constraint will hold after appliance of the fix), the language cannot express arbitrary complex fixes and constraints, and the authors admit that it might yet be very difficult to devise the actual constraint/fix expression. Nentwich et al. [12] evaluate first order logic constraints in order to derive the correct and complete set of all possible fixes. Egyed's "incremental inconsistency fixing" [9], in contrast, is independent of the checking language used, as it instruments the modeling infrastructure to trace all model elements a constraint accesses. The approach leverages the fact that the set of elements accessed by an inconsistent constraint constitute the correct and complete set of *locations* that can potentially fix the inconsistency [9].

Both Nentwich's and Egyed's approach are promising candidates to be integrated into our framework. As the former makes strict demands on the constraint checking language, we decided to initially adapt and extend only Egyed's approach to our needs.

### D. Declarative Fixing Mechanism

Our declarative approach is based on the approach of Egyed [9]: As well, we use an instrumented modeling infrastructure to traces all model accesses. However, Egyed's approach has two drawbacks when it comes to fixing. First, the complete set of fix locations per constraint yet is rather large, in [13], for example, it has more than 10 entries on average, which hampers usability when having many constraints. Second, the approach does not concern how to derive actual fixes ("fix operations") from the fix locations when a fix requires complex model changes or additional user input.[5]

In order to relieve the user from too many automatically-derived inconsistency fixes, we combine model access tracing with a *declarative marker mechanism* integrated into the constraint language. This way, the constraint developer explicitly can mark elements for which fixes shall be derived. (This means that we dismiss the property of completeness in favor of keeping track, while we still can guarantee the correctness of each fix location suggested.)

The declarative marker mechanism is implemented via the method `suggest()`. It can be called within a constraint on any data type or object. It is an identity function with exactly one side effect: it informs the tracing facility that the model element last accessed is a candidate for fixing.[6]

```
Object obj.suggest(
  [Object defaultValue [, String descr ]])
```

[5]In [13], the author shows how to performantly compute those fixes that can be derived *purely* from the contents of the current model without other user input.

[6]Therefore, the checking language must not reorder function execution for optimization. For Xpand Check, this is the case.

Concrete fix actions are then derived automatically depending on the object type and are also presented in the Fixing GUI (cf. Section IV-B). For primitive data types (String, double, int, boolean) either a user dialog is displayed or the `defaultValue` is chosen. In case the object type is a reference, a dialog can be displayed to change the reference to another object of compatible type. As we base on the incremental checking and fixing approach of [9], the complexity of our approach is as well linear (both in memory and computation cost). Most of the total overhead of only 6 percent is caused by the instrumentation of our modeling infrastructure (Eclipse Ecore) for tracing (cf. Section VI for details).

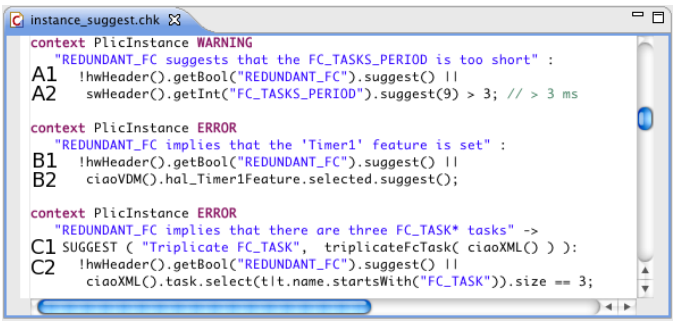


Fig. 4. The three exemplary constraints (already seen in Figure 3), each comprising two suggestion fixes. Suggestion A1 (as well as B1 and C2) will suggest to toggle the value of the `REDUNDANT_FC` define. Suggestions A2 sets the `FC_TASKS_PERIOD` define to a value queried interactively from the user (default value: 9 ms). Suggestion B2 toggles the boolean feature `hal_Timer1`, whereas C1, the only imperative fix, suggests a triplication of the flight control tasks instantiated in the CiAO XML file calling an externally defined Xtend function `triplicateFcTask()`.

### E. Example Constraint Fixes

Figure 4 shows the constraints of Figure 3 with suggestions attached (using both `suggest()` functions and `SUGGEST` annotations). For each constraint that does not hold, the user will be presented the set of textual descriptions of the attached suggestions (both declarative and imperative ones) in the Fixing GUI. On selection, the corresponding fix action is performed.

`SUGGEST` annotation fixes are more complex to write, as they require the implementation of a separate Xtend function (e.g., 15 LOC in case of the `triplicateFcTask()` function). However, the fix can be arbitrarily complex. Using `suggest()` functions, in contrast, it is not possible to encode complex domain knowledge. They are, however, very concise and keep the effort for defining a fix minimal (only one function call on the element which shall be fixed).

## V. Inconsistency Fixing Process

Considering the round-trip and fixing mechanisms integrated, a process for applying the fixes should consider two issues. First, in particular when complex imperative fixes are applied on a model, the user must keep in control and keep track of the actual changes. Second, the process needs to deal with possible *semantic inconsistencies* that our round-trip approach for feature models and KConfig does not check for. As described in Section III-B, our current converter approach ignores the constraints defined within feature models or KConfig files.

### A. Inconsistency Fixing Process

Figure 5 depicts the process for fixing inconsistencies. It addresses the two mentioned issues by including explicit steps at which the user can review the fixed models and by implementing additional semantic checking *after* the fixed model has been serialized to a file. The process splits up into two major parts: *applying fixes* (1 to 3) and *semantic checking & resolving* (4 to 7).

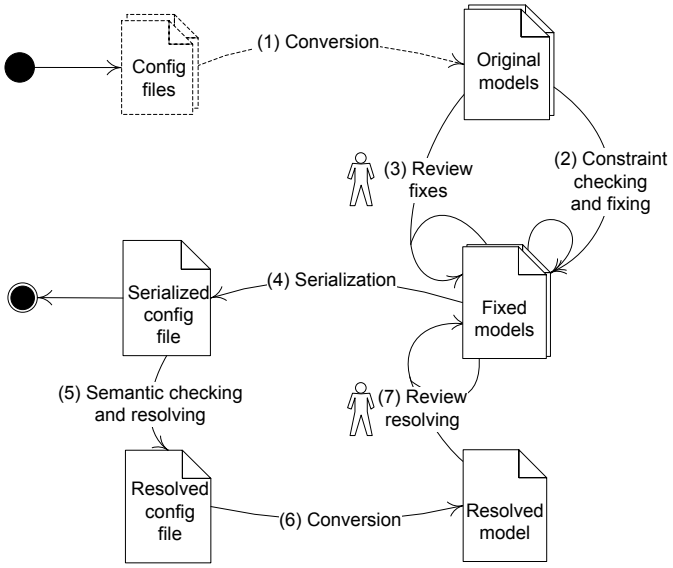


Fig. 5. Process applied to fix inconsistencies.

**Applying Fixes (1 to 3).** In the beginning (1), the inconsistency fixing framework uses the appropriate converters for each configuration file to create a model from it (*original models*). Then, the models are inconsistency checked and the user may repeatedly apply fixes using the Fixing GUI to remove configuration inconsistencies (2), resulting in the *fixed models*. As the user might want to review how the applied, possibly complex fixes have *actually* affected the original model (3), the framework incorporates a facility to show the difference on model level to the user (Section V-B will give further details).

**Semantic Checking & Resolving (4 to 7).** When finished with fix application, the user may initiate semantic checking & resolving for each model. First, the model is serialized (4) to its original file format. Then, file-type–specific semantic checks and resolvers may be applied on the generated configuration file (5) (Section V-C will provide details). The result of resolving is a *resolved configuration file*, which is converted back to a model representation (6), called *resolved model*. Its differences with respect to the previous *manually resolved model* can again be reviewed by the user (7), who may decide to accept or discard each change, resulting in a new version of the *fixed model*. The cycle may be repeated until a semantically valid configuration file is achieved or the user cancels resolving.

### B. User Control and Guidance

In order to give the user control and guidance over the applied fixes, we implemented two mechanisms. First, as already described previously, the Fixing GUI presented to the user shows the textual descriptions attached to each fix to provide information about its impact. Second, we implemented a Reviewing GUI. Similarly to a "diff"-viewer on text level, it shows the differences between the original model and the fixed model on model level and is also able to retract already applied changes.[7] This reviewing mechanism is particularly useful when complex fixes have been applied, for which the outcome does not become clear through its textual description.

### C. Semantic Checking and Resolving

Even in case serialization succeeds, the resulting file can be invalid with regard to the constraints imposed by the specification file (e.g., the constraints defined in the feature model, or in a KConfig file). In those cases, we *semantically validate* the files generated from the fixed models using the original tools, pure::variants and the KConfig command line tool, respectively. Interestingly, both provide validation checkers and autoresolvers: Pure::variants via a Java API working with a Prolog engine, KConfig via hand-crafted algorithms. In case a validation checker fails, the user can choose to apply the respective autoresolvers. Although we cannot provide the user with control over *how* an autoresolver reconstitutes validity, we can present the user with the applied changes in a concise format. Therefore, we again leverage the Reviewing GUI, which shows the differences concisely on model level.

## VI. Evaluation

In this section we shortly describe a user study that indicates a configuration time speedup when using our process. Furthermore, we will report on the performance impact of our constraint language extension for declarative fixing.

### A. Process Evaluation

We performed an experimental user study to test whether our user-guided fixing process speeds up inconsistency fixing compared to manually adaptation of configuration files.

Eight participants had to fix inconsistencies in an ill-configured quadrotor system, which, due to its configuration complexity, constitutes a suitable evaluation subject.[8] About 30 changes in two header files, a feature model configuration, and an XML configuration file were required in order to restore consistency. The participants were split up into two roughly equally-skilled groups regarding the system's software; none had used our fixing framework before. Group 1 first performed the changes by *manually adapting configuration files* using the respective Eclipse editors for text, XML and pure::variants feature model configurations (Task A). Therefore, detailed textual hints what to edit to restore consistency were displayed in the Eclipse problems view for each constraint evaluating to false, to avoid pauses for reflection. Then, Group 1 performed the adaptations by using our *process for user-guided fixing of inconsistencies* (Task B). Group 2 performed the same two tasks, but in the reverse order, to sort out learning effects.

| Pt. | Config. Know-how | | Effort in minutes | | | |
|---|---|---|---|---|---|---|
| | | | Task A | Task B | | |
| | | | Manual | Tool-supported | | |
| | Copter | OS | Total | Total | Apply | Serialize |
| **Group 1** | | | | | | |
| 1 | o | o | 17:07 | 4:33 | 2:20 | 2:13 |
| 2 | + | o | 15:33 | 4:10 | 2:08 | 2:02 |
| 3 | o | + | 14:06 | 5:13 | 2:51 | 2:22 |
| 4 | + | + | 15:26 | 4:39 | 2:30 | 2:09 |
| **Group 2** | | | | | | |
| 5 | o | o | 18:00 | 6:47 | 4:11 | 2:36 |
| 6 | + | o | 12:12 | 6:09 | 3:59 | 2:10 |
| 7 | o | + | 9:30 | 6:00 | 3:36 | 2:24 |
| 8 | + | + | 10:15 | 5:25 | 3:15 | 2:10 |
| Average (total) | | | **14:01** | **5:21** | 3:06 | 2:15 |
| Median (total) | | | **14:46** | **5:19** | 3:03 | 2:11 |

Pt.: Participant | Config.: Configuration
Apply: Time spent in "applying fixes" phase
Serialize: Time spent in "semantic checking & resolving" phase
+: Configuration experience | o : No configuration experience

TABLE II
Time effort for fixing inconsistencies.

Table II shows the results of the study performed. The medial participant was able to perform the configuration task in only 5 minutes and 19 seconds when using our fixing tooling, compared to 14 minutes and 46 seconds when performing the task manually. This corresponds to a relative time saving of 64 percent. All participants considerably profited from our tooling.

Our fixing tooling works in two phases. In the first phase, the user chooses the fixes from the GUI, possibly entering some integer or string values (*Apply* phase in Table II). The participants performed this task in about three minutes. After applying the changes, each participant spent another two minutes and eleven seconds in the semantic checking & resolving phase (*Serialize* phase in Table II). Most of this time has been consumed for validating and autoresolving the feature model configuration; about one minute was spent in the corresponding validation and autoresolving Java API functions. Overall, we say that the evaluation gives evidence that automatic support for fixing inconsistencies by encoding configuration knowledge as constraints and fixes, as provided by our process, can achieve considerable time savings.

**Threats to Validity.** The experimental setup examined a nontrivial configuration problem of a system (80 person months) and the participants had a strong computer science background with varying experience (0 to 5 years) with the involved systems, so that relevant industrial setups are covered. The editors used for manual adaptation were the IDE's standard editors for the respective file types, how they are likely to be used during usual configuration tasks. By observation, we could affirm that the textual hints given for manual adaptations were clear enough, so that understanding problems did not derogate the results. However, as a drawback, the measured improvements are only transferable to inconsistencies for which a definite set of change operations fixes can be specified in advance.

[7] This *diff&merge dialog* has been implemented using EMF Compare: http://www.eclipse.org/emft/projects/compare/.

[8] The instructions given to the to participants can be downloaded from http://www4.cs.fau.de/~elsner/instr.pdf.

Furthermore, the presented figures were produced *before* we integrated incremental consistency checking and fixing; all constraints (not only those that could possibly change) were checked after each fixing cycle (Figure 1, step 2). However, as both the absolute and relative performance impact of instrumentation is moderate—and mostly even beneficiary, as the set of constraints to reevaluate after a fix is minimized (cf. subsequent section)—the prospective speed up will not be significantly different.

### B. Fixing Extension Performance Impact

Although our constraint language extension relies on a modeling infrastructure instrumentation that traces all accesses to model elements, its performance impact is very moderate. We executed the full set of constraints used for flight control redundancy ($\approx$ 30) on a laptop with Core 2 Duo 2.4 GHz a hundred times without and with tracing enabled. The average execution time increased only 6 percent, from 513 ms (min: 425 ms, max: 853 ms) to 544 ms (min: 433 ms, max: 938 ms), to log the 3,000 involved model elements. However, when incremental consistency checking is activated, checking the full set of constraint is only necessary once. Afterwards, only constraints actually affected by a fix are reevaluated. In our quadrotor helicopter case study, for example, the average fix affects the reevaluation of less than three constraints, which then takes only around 150 ms in the average case (min: 109 ms, max: 255 ms), which is only 29 percent of the average execution time for all constraints.

## VII. Discussion

In the following, we discuss the reliability of the developed round-trip mechanisms and fixing process termination.

Developing bidirectional transformation is not without pitfalls. Nevertheless, the use of mature frameworks and common development practices very much lowers the problems usually associated with it. For the round-trip mechanisms of Ecore DSMs, XText DSLs, and XMLSchema XML, we could leverage the widely-used and mature EMF framework. Due to the restrictions we impose on CPP header files, and the simple structure of Java property files, the implementation of their converters and serializers has been decently simple as well. For complex file types, such as for feature models and KConfig files, finally, we use a pragmatic approach that only aims at syntactical correctness; we shift the heavy lifting of semantic checking to the respective original tooling.

Our approach supports imperatively programmed fixes on models of arbitrary metamodels. This has the drawback that we cannot determine the impact of a fix until it is actually applied. Hence, we are not able to guarantee that a fix will not invalidate other constraints, nor can we guarantee that a totally consistent state can be reached at all. A way to solve this problem would be to limit the allowed models (e.g., only finite number of states) and/or the constraint language (e.g., to boolean or first order logic), so that SAT solvers or CSP could be used. This, however, is a step we did not want to take in order not to limit the applicability of our approach.

## VIII. Conclusion

In this paper, we have presented the design of a framework that enables to specify and to apply inconsistency fixes on configuration files of various types. Modeling has proven to be a very suitable pivot technology; by leveraging existing model-based frameworks and common development techniques, if was feasible to develop reliable round-trip mechanisms of reasonable quality for various file types with reasonable effort. Furthermore, it enabled us to adapt and extend previous research in model-based inconsistency fixing by adding a marker mechanism that limits the number of suggested fix locations. Finally, our framework incorporates a process to deal with complex fixes and semantic inconsistencies: For the former, we provide the user with a GUI mechanism to review and control the actually applied changes on model level, for the latter, we shift the heavy lifting of checking file–type-internal constraints (e.g., constraints in feature models) to the respective original tool, which is best suited for this purpose.

## References


[1] C. Elsner, P. Ulbrich, D. Lohmann, and W. Schröder-Preikschat, "Consistent product line configuration across file type and product line boundaries," in *14th Software Product Line Conf. (SPLC '10)*, ser. LNCS, K. Kang, Ed., vol. 6287. Springer, Sep. 2010, pp. 181–195.

[2] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[3] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[4] A. Bohannon, B. C. Pierce, and J. A. Vaughan, "Relational lenses: a language for updatable views," in *PODS '06: 25th ACM SIGMOD SIGACT SIGART Symp. on Principles of database systems*. New York, NY, USA: ACM, 2006, pp. 338–347.

[5] H. Bruneliere, J. Cabot, C. Clasen, F. Jouault, and J. Bezivin, "Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools," in *6th European Conf. on Modelling Foundations and Applications (ECMFA '10)*. Heidelberg, Germany: Springer, 2010.

[6] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *Int. Conf. on Model Transformations (ICMT '09)*, ser. LNCS, vol. 5563. Heidelberg, Germany: Springer, 2009, pp. 260–283.

[7] A. Schürr and F. Klar, "15 years of tripple graph grammars – research challenges, new contributions, open problems," in *4th Int. Conf. on Graph Transformation (ICGT '08)*, ser. LNCS, vol. 5214. Heidelberg, Germany: Springer, 2008, pp. 411–425.

[8] M. Stephan and M. Antkiewicz, "Ecore.fmp: A tool for editing and instantiating class models as feature models," University of Waterloo, 200 University Avenue West Waterloo, Ontario, Canada, Tech. Rep., Aug. 2008.

[9] A. Egyed, "Fixing inconsistencies in UML design models," in *29th Int. Conf. on Software Engineering (ICSE '07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 292–301.

[10] "Eclipse XPand homepage," http://www.eclipse.org/modeling/m2t/?project=xpand, visited 2011-02-02. [Online]. Available: http://www.eclipse.org/modeling/m2t/?project=xpand

[11] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei, "Supporting automatic model inconsistency fixing," in *ESEC/FSE '09:the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 315–324.

[12] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency management with repair actions," in *25th Int. Conf. on Software Engineering (ICSE '03)*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 455–464.

[13] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *23th IEEE Int. Conf. on Automated Software Engineering (ASE '08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 99–108.