# Automated Generation of Consistency-Achieving Model Editors

Patrick Neubauer, Robert Bill, Tanja Mayerhofer, and Manuel Wimmer
Business Informatics Group, TU Wien, Austria
{neubauer, bill, mayerhofer, wimmer}@big.tuwien.ac.at

***Abstract*—The advances of domain-specific modeling languages (DSMLs) and their editors created with modern language workbenches, have convinced domain experts of applying them as important and powerful means in their daily endeavors. Despite the fact that such editors are proficient in retaining syntactical model correctness, they present major shortages in mastering the preservation of consistency in models with elaborated language-specific constraints which require language engineers to manually implement sophisticated editing capabilities. Consequently, there is a demand for automating procedures to support editor users in both comprehending as well as resolving consistency violations. In this paper, we present an approach to automate the generation of advanced editing support for DSMLs offering automated validation, content-assist, and quick fix capabilities beyond those created by state-of-the-art language workbenches that help domain experts in retaining and achieving the consistency of models. For validation, we show potential error causes for violated constraints, instead of only the context in which constraints are violated. The state-space explosion problem is mitigated by our approach resolving constraint violations by increasing the neighborhood scope in a three-stage process, seeking constraint repair solutions presented as quick fixes to the editor user. We illustrate and provide an initial evaluation of our approach based on an Xtext-based DSML for modeling service clusters.**

***Index Terms*—Domain Specific Modeling Languages, Model Driven Engineering, Search-based Software Engineering, Advanced Editor Support**

## I. Introduction

Domain-specific modeling languages (DSMLs) [1] encode the expertise of domain experts [2], e.g., hospital process managers and mechatronics engineers, and are, hence, languages designed for a specific class of problems allowing domain experts to describe their problem on a higher level of abstraction than possible with general-purpose modeling languages (GPMLs) [3]. DSMLs are known to leverage domain expertise and improve usability, comprehensibility, and maintainability compared to alternatives. On the one hand the benefits of DSMLs have been recognized, but on the other hand the development of advanced DSML implementations still requires extensive manual effort and the combined knowledge of domain and language engineers resulting in slow industrial adoption. Therefore, to advance the state-of-the-art in maintenance and evolution of languages and systems created with them, our goal is to significantly reduce the cost associated with the creation and thus the adoption of advanced DSML implementations required by currently available approaches [4].

Model Driven Language Engineering (MDLE) frameworks (a.k.a. language workbenches) such as Xtext [5], ease the out-of-the-box creation of powerful DSML implementations, including associated editors [4]. Typically, an MDLE framework allows creating a language either by developing a metamodel representing the language's abstract syntax, which is subsequently used to generate a DSML implementation including language grammar, or by developing a language grammar, which is automatically used to generate an associated metamodel. Either way, generated DSML implementations require extensive manual implementation for enabling advanced validation, content-assist, and quick fix capabilities. For example, in case a metamodel containing formally specified consistency constraints, such as restricting the `speed` attribute of a `Server` element to be represented by an Integer greater than `0`, is used to derive a DSML editor, it does not take into account such restrictions for displaying an appropriate validation result. In more detail, it highlights the entire `Server` element as erroneous and does not provide any quick fix to solve the violated constraint. Further, to take formal constraint specifications into account, the language developer has to manually implement validation, content-assist, and quick fixes using a general-purpose programming language, such as Java, on top of the generated DSML editor.

On one side, advanced editing capabilities decrease the effort to create and modify instances of a language (i.e., models) and therefore increase the productivity of language use. On the other side, they significantly increase the development effort for DSML editors.

Moreover, models created with DSMLs as well as the DSMLs themselves evolve [6], [7]. Each time a metamodel or its associated formal constraint specifications are adapted, handwritten implementations for model validation, content-assist, and quick fixes need to be accustomed as well.

In this work, we present an approach to automatically generate advanced validation, content-assist, and quick fix capabilities for DSMLs based on formal constraint specifications residing in language metamodels. Hence, the approach exploits language metamodels equipped with formal constraint specifications to achieve DSML implementations that include the desired capabilities, thus reducing initial and running costs for creating and maintaining editors which allow to efficiently create and maintain models.

Formal constraint specifications may be added to the language metamodels in terms of Object Constraint Language

978-1-5090-5501-2/17 © 2017 IEEE

SANER 2017, Klagenfurt, Austria
Main Research

(OCL) [8] constraints and Ecore annotations added to metamodel elements. In our approach, we extract OCL constraints and Ecore annotations from metamodels and analyze them to create advanced DSML editors. For doing so, we had to face several challenges: ($i$) narrow the scope of infinite possible ways to repair a given set of inconsistencies to a practically applicable level, e.g., as done by Nentwich et al. in [9], and ($ii$) measure the impact of an identified inconsistency repair solution as it may introduce new inconsistencies and therefore may be counterproductive [10].

We hypothesize that, by applying our approach, a DSML implementation is created, which enables to ($i$) accelerate the creation and editing of DSML models, ($ii$) build DSML models with fewer errors, and hence ($iii$) improve the overall productivity and quality of DSML creation by language engineers and DSML usage by editor users.

Our vision for the future is to completely automate the evolution of existing languages, e.g., the evolution of markup languages, such as XML Schema languages, to advanced DSMLs providing instance-level backward compatibility [11]. Hence, in this paper, we focus on the particular aspect of increasing the quality and usability of DSML editors by extracting formal specifications from the metamodel that is part of the DSML as well as its editor.

A realization and evaluation of our approach has been made available in form of the IntellEdit framework[1] as well as its application on a DSML for modeling service clusters.

In the next sections we describe ($i$) the background by providing an overview of appearing concepts, ($ii$) related work, ($iii$) a motivating example, ($iv$) the generic approach as well as our concrete implementation, ($v$) the evaluation of our implementation of a language for modeling service clusters, ($vi$) a detailed comparison of our approach with existing language workbenches, and ($vii$) the conclusion including a presentation of future work.

## II. Background

This section describes key concepts behind our approach. Gray et al. define domain-specific modeling as being tightly coupled to a language that is by definition linked to the domain over which it is valid [2]. Consequently a domain-specific (modeling) language (DS(M)L) [12] is a language tailored to a specific application.

DSMLs are based on MDE [13] technologies that formalize application structure, behavior, and requirements within particular domains using metamodels defining relationships among concepts in a domain as well as key semantics and constraints associated with such concepts [13]. Therefore, the development of DSMLs, which is considered challenging, requires developers with domain expertise as well as language engineering expertise from which few of them have both [12]. Moreover, DSMLs are intended to be used by developers to build applications expressing design intent declaratively rather than imperatively and hence require significant customization before they can be applied in practice [14].

To support the development of DSMLs, MDLE frameworks, such as Xtext [5], emerged, enabling language designers to ease language development by leveraging advances in editor technology of mainstream IDEs. Such MDLE frameworks automate, for instance, the creation of language specific parsers, serializers, and editors providing basic syntax highlighting, content-assist, folding, jump-to-declaration, and reverse reference lookup across multiple files. However, their validation, content-assist, and quick fix capabilities are limited, not only due to the unavailable information but also due to the capacity in which the available information is exploited. For example, validators typically highlight an entire class as invalid instead of the actual feature violating a constraint, which makes identification and resolving of errors more difficult and time-consuming.

On one hand and as found in our earlier work [15], the generation of language grammar from metamodels through MDE techniques and MDLE frameworks still suffers several limitations, such as the ability to store values for data type instances, and hence require extensive manual customization and extension by language engineers. On the other hand, if a metamodel is derived from a language grammar, that metamodel is not supplied with formal constraints. In particular, we do not know of any approach which allows translating constraints defined in e.g. attribute grammars to metamodel constraints. However, formal constraints have become key components in MDE for expressing different kinds of (meta)model queries and specification and manipulation requirements. Moreover, in the technological space of Grammarware, it is not feasible to construct and maintain complex formal constraints.

As a textual formal constraint language, the standardized Object Constraint Language (OCL) [8] is widely used to enhance modeling languages with unambiguous and precise constraints. OCL constraints are captured as invariants using a typed, declarative and side-effect free specification language supporting first-order predicate logic offering navigation and (meta)model querying facilities.

However, while OCL represents the most used textual formal specification language in the area of MDE, several shortcomings have been identified [16]. These include ($i$) poor support for user feedback, ($ii$) no support for warnings, ($iii$) no support for dependent constraints, ($iv$) limited flexibility in context definition, and ($v$) no support for repairing invariants.

## III. Related Work

Egyed et al. and Reder et al. [17], [18], [19] present an approach to assist editor users in fixing inconsistencies in UML models by generating a set of concrete changes as well as their impact on consistency rules. In [18] and [19] they focus on the cause of inconsistencies by analyzing the consistency rules as well as their behavior during the validation, and present validation as well as repair in form of linearly growing trees. They evaluated the scalability of their approach by applying it to UML models and OCL rules. They were able to compute

[1] A ready-to-use virtual machine image and Eclipse instance of the IntellEdit framework as well as evaluation results can be retrieved online from http://intelledit.big.tuwien.ac.at.

validation and repair trees in the millisecond-range, which indicates that the application of the tree data structure may be highly appropriate for the problem at hand. Our approach differs in the way it searches for complete repairs but is similar with respect to basic repair rules such as the insertion of a reference.

In [17] they found that the performance of applying the brute force technique to generate repair solutions is too low and hence decided to add manual specifications in form of value generation functions and (back)pointer specifications. Compared to our approach, they follow the same motivation for automating the generation of repairs for constraints that otherwise require significant manual effort. However, our approach avoids the manual burden on the language engineer in the manual specification of value generation functions and (back)pointer references by adopting Search-based Software Engineering (SBSE) [20] techniques instead. SBSE entails the application of optimization techniques, originating from metaheuristic computation and operations research. Moreover, we hypothesize that the language engineer may not be able to feasibly state all possible value generation functions and (back)pointer specifications and hence unintentionally limits the generation of potentially favorable quick fix solutions. Furthermore, their approach does not support ($i$) the generation of fixes of inconsistencies whose resolution does not require the introduction of new model classes or attributes and ($ii$) the generation of resolutions that involve changes in multiple locations at a time.

Hegedüs et al. [21] present an approach to automate the generation of quick fixes for DSMLs by taking a set of constraints and model manipulation policies as input. Their repair solutions are realized as a sequence of operations, which are computed by employing state space exploration techniques, targeted to decrease the number of inconsistencies. Compared to our approach, they also look for local and global quick fix solutions. However, they do not take content-assist as well as the overall validity of the model into account. Moreover, they employ graph patterns to capture inconsistency rules and graph transformation rules for repair operations instead of textual formal constraints and SBSE techniques.

Silva et al. [22] describe a method for the generation of repair plans for an inconsistent model by configuring the explored search space to antagonize the underlying characteristic of the problem, i.e., the infinite amount of possible repair solutions, at hand. Compared to our approach, their approach restricts the search space exploration by ($i$) limiting the generation of repair solutions in general, ($ii$) the applicability to every model at hand, and ($iii$) the requirement to manually create inconsistency detection rules. Therefore the quality of generated results will suffer and the burden on the language engineer, in terms of language implementation and maintenance, is increased due to the manual construction of inconsistency detection rules. In our approach, we asynchronously look for solutions during editor execution and thus can potentially find solutions to resolve inconsistencies with a higher quality compared to results found in a limited amount of time.

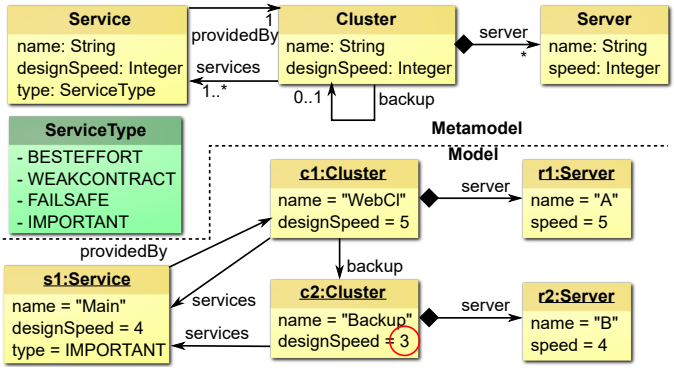


Fig. 1. Metamodel and model of the motivating example.

Ali et al. [23] and Semerth et al. [24] generate models which fulfill OCL constraints by optimization and reasoning, respectively. However, when compared to our approach, they only consider single-objective optimization, i.e., generated test models need to fulfill all OCL constraints which only considers the model state, and not how to get there. Therefore, they neglect the power of multi-objectiveness [25], which is required for establishing quick fix solutions for edit operations based on multiple conflicting objectives, e.g., considering both solution validity as well as solution length as optimization objectives.

## IV. Motivating Example

This section introduces our motivating example, represented by a DSML for modeling service clusters, which is referred to in the following sections. We do so by describing its metamodel including formal constraints, which are employed by our approach for creating advanced editor support.

**Metamodel.** The DSML metamodel (cf. Fig. 1 upper part) is composed of the classes Cluster, Service, and Server. A Service is defined in terms of *designSpeed*, i.e., an Integer representing the speed for which a service is intended to be used, *type*, which may either be *BESTEFFORT*, *WEAKCONTRACT*, *FAILSAFE*, or *IMPORTANT*, and has to be provided by exactly one cluster. A Cluster has a *designSpeed*, i.e., an Integer representing the speed for which a cluster is intended to be used, as well as one or multiple containing servers. Moreover, a cluster is associated with one or more services and up to one backup cluster. Finally, each Server has a *speed* that it provides.

**Formal Constraints.** Formal constraints are defined for servers, clusters, and services (cf. Listing 1). For example, a Service is constrained by the type of service that it provides. In detail, it has to either have a *BESTEFFORT* service type with no further restrictions or, in case of a *WEAKCONTRACT* service type, its associated cluster has to be designed for a speed that is equal or greater than the designed speed of the service itself. A *FAILSAVE* service type means that some backup cluster has to be provided. The *IMPORTANT* service type further extends the restrictions associated with the *FAILSAFE* service type by requiring the designed speed

```
– Service
invariant speedFulfilled: type =
  ServiceType::BESTEFFORT or (
    designSpeed <= providedBy.designSpeed and
      (if type = ServiceType::IMPORTANT then
        designSpeed <= providedBy.backup.designSpeed
      else type = ServiceType::WEAKCONTRACT or
       providedBy.backup <> null endif));
```

Listing 1. Selected constraints for the motivating example.

of the associated backup cluster to be equal or greater than the designed speed of the service itself.

**Example Model.** We consider the example as shown in the lower part of Fig. 1. We have a single *important* service with speed *4* which is provided by the Cluster *WebCl*. This cluster has a low-speed backup which is not sufficient (cf. speedFulfilled) since its design speed is lower than the speed required by the main service. In the following, we will show how this example will be handled by our extended validation, content-assist and quick fix providers.

## V. The Approach

Our approach has been realized in the *IntellEdit framework*. The functionality of our framework is applied both during the generation of *Advanced DSML Editors* (cf. Fig. 2) as well as during their execution (cf. Fig. 3).

In the following, we illustrate our framework in terms of the generation and execution of advanced DSML editors and detail on our generic approach for the validation of constraints, the production of content-assist suggestions, as well as the production of quick fix solutions.

### A. Generation of Advanced DSML Editors with IntellEdit

*IntellEdit* is a Java framework build on top of the Eclipse Modeling Framework (EMF) [26], Xtext [5], Multiobjective Evolutionary Algorithm (MOEA) framework [27], and OCL [8]. It has been specifically created to realize our intentions of leveraging the advantages of comprehensive language definitions. Therefore, the language engineer first applies the EMF facilities, to specify a DSML *Metamodel* based on Ecore, i.e., a common standard for metamodeling. Next, the language engineer augments the *Metamodel* with *Textual Formal Constraints* (cf. Fig. 2) in terms of OCL constraints that enable to specify, e.g., restrictions, such as, equations having equal values on both sides. Thus, for a model to be valid, all constraints defined in its associated metamodel must be fulfilled.

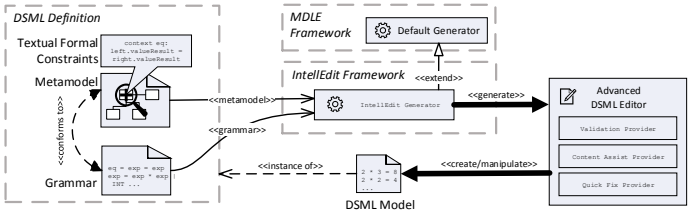


Fig. 2. IntellEdit Framework: Adv. DSML Editor Generation.

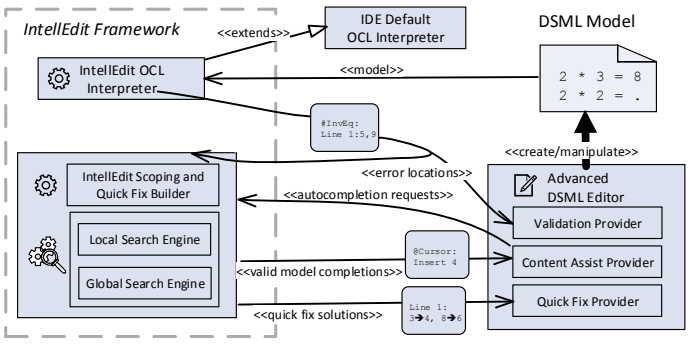


Fig. 3. IntellEdit Framework: Adv. DSML Editor Usage.

Having completed the *DSML Definition*, i.e., representing the language definition, our framework generates a conforming *Advanced DSML Editor*, which provides a customized *Validation Provider*, *Content Assist Provider*, and *Quick Fix Provider*. To do so, we apply both EMF default facilities as well as the *IntellEdit Generator*. Moreover, by separating classes used for validation, content-assist, and quick fix solutions from other editing classes, *IntellEdit* enables straightforward injection into automatically generated files.

During the execution of the generated *Advanced DSML Editor*, which is used to construct and manipulate models that represent instances of the DSML definition, the editor issues run-time requests to our framework. Subsequently, *IntellEdit*, which employs the MOEA framework during this step, establishes advanced validation, content assist, and quick fix results and replies them to the *Advanced DSML Editor*.

### B. Generic Approach for Advanced Constraint Validation

Editors that are generated by state-of-the-art tools tend to visualize single inconsistencies in terms of the context in which the violation occurs instead of a more exact location, e.g., the union of all minimal error causes. This is problematic, as the language modeler is presented with imprecise information on the correctness and incorrectness of the model. Proper visualization of error causes can contribute to a better modeling process that avoids subsequent errors [19].

***Definition of Change Action Requirement Types.*** In our approach, we consider change action requirement types that are constructed and evaluated in evaluation trees. For the purpose of acquiring potentially relevant error locations, we perform a runtime-analysis of the expression evaluation, by comparing *expected result* with *actual result*, and return locations where deviations occur. To do so, we store the expected result value as one or more of the following eleven change action requirement types that provide a good basis for finding some correcting changes with decent speed in the first two layers of our search algorithm. These change action requirement types can be seen as a specialization of the general model constraints to basic constraints on individual model element properties. First, different types of expected results, generally corresponding to a simple set of conditions or single values conditions, are represented by equal($v$) and different($v$) and denote that expressions should have the same value or a different value than the given one. Secondly, *true* and *false*

correspond to equal(*true*) and equal(*false*), respectively. Next, the expected results increase($v$) and increaseexcl($v$) as well as decrease($v$) and decreaseexcl($v$) for Integers denote that a value should be at least or at most a specific value. Moreover, contains($v$) and excludes($v$) denote that an expected set of results should contain or should not contain the value $v$. Likewise, the anticipated size of an expected set of results is indicated by minsize($c$) and maxsize($c$). Finally, the definition of change on the expected result indicates that the criteria of a particular expression are not known.

Moreover, the deviation of many typical OCL constraint evaluation can be encoded using these basic requirement types, including setting properties, various collection operations, simple inequalities, Boolean operators and single-parameter functions with a computable inverse set.

```
1  name: fixgen
2  input: f ⊆ Car, (e, v) ∈ {(Expression, Result)}
3  output: SURE m : (Expression, Result) → Car // result fixes
4    f ← f \ check(f, v)
5    m ← (e, v) ↦ f else ∀a, b : (a, b) ↦ {}
6    u ← f //unhandled fixes
7    for p ∈ availableGenerators(e.type())
8      (h, m_s) ← p.get(e, v, f)
9      u ← u \ h
10     m ← (a, b) ↦ (m(a, b) ∪ m_s(a, b))
11   if u ≠ ∅
12       for (e_s, v_s) ∈ subresults(e, v)
13           m ← (a, b) ↦ (m(a, b) ∪ ((e_s, v_s), Change))
14   for (e_s, v_s) ∈ subresults(e, v)
15       m_s ← fixgen(m(e_s, v_s), (e_s, v_s)
16       m ← (a, b) ↦ (m(a, b) ∪ m_s(a, b))
```

Listing 2. Change Action Requirement generation.

***Evaluation of Change Action Requirement Types.*** The basic generation of change action requirements *Car* as defined above is described in Listing 2. In case the evaluation of a change action requirement yields a positive result, i.e., the requirement is fulfilled, changes of existing sub-evaluations are not required and hence, propagation is skipped. On one hand, expression types for which expected results are known may be specified accordingly using a change action requirement generator. On the other hand, expression types for which expected results are not known have their most general expected result, i.e., change, propagated to all sub-expressions.

There are several change action requirement generators which can be used in our approach. For operations with a finite operation table and a certain target value, all possibly expected results of a sub-expression are determined by looking at the operation table, i.e., set of operation results for input values.

```
input f : (x_1, ..., x_n) → y Function, c_1, ..., c_n current assignment,
       vexpected value
output r : P((x_1, ..., x_n)) //set−minimal changes
  b = f^{-1}(v)
  b_h ← {{(i, k_i)|k_i ≠ x_i}|(k_1, ..., k_n) ∈ b}
  b_h ← b_h \ {A ∈ b_h, ∃B ∈ b_h : B ⊊ A}
  r = {(r_1, ..., r_n)|A ∈ b_h, r_i = if ∃(i, z) ∈ A then z else x_i end}
```

Listing 3. Preparation of change action requirements generator operation for finite functions.

In detail, assuming an operation $f(x_1, ..., x_n) = v$, with current sub-expression evaluations $y_1, ..., y_n$ for all sub-expressions and expected value $v$, a suitable change $\{y_k \rightarrow y'_k, ...\}$ fulfills $f(y_1, ..., y'_k, ...y_n) = v$. The computation of all suitable set-minimal changes, which is computationally intensive, is performed once for every operation on application startup and is depicted in Listing 3. Listing 4 shows an algorithm which actually applies such a fix to an existing evaluation. It handles all $\mathrm{equal}(x)$ requirements by trying to find base values which make the function return the expected value $x$ and tries to recursively apply these change action requirements.

TABLE I
REDUCED TABLE FOR MAKING THE OR EXPRESSION TRUE.

| OR | A:false | A:true |
|---|---|---|
| **B:false** | A → *true*, B → *true* | - |
| **B:true** | - | - |

Table I shows a simple example of such a table for making *OR true*. Only in the case of both *A* and *B* being false, propagation is triggered. In that case, both *A* and *B* may become true.

Further, for the Boolean conditions $v_1{=}v_2$ and $v_1{\neq}v_2$ and expected values true and false, the expected values can be propagated to equal($v_2$) and equal($v_1$) for the first and second sub-expression, i.e., $v_1{=}|{\neq}v_2$ :=*true*|*false*, respectively. For the other combinations, different can be applied. Similarly, increase and decrease are utilized for the inequalities $<$ and $>$, respectively. Likewise, for inclusion and exclusion in set memberships of expected Boolean values, contains and excludes may be used, respectively. Furthermore, expected Integer values of set sizes are converted with minsize and maxsize. Inclusions and exclusions of set selection operators, i.e., select, $\{x \in S|\mathsf{cond}(x)\}$, are mapped to ($i$) inclusions and exclusions of the source set $S$ and ($ii$) an expected value of *true* for every object that should be contained in the set and *false* for every object that should not be contained in the set. Set collections, i.e., collect, $\bigcup_{y \in S} x \in f(y)$, propagate their excluded elements both to $f(y)$ and to remove $y$ from $S$ where an unwanted value is calculated and their included elements yield an inclusion in $f(y)$ as well as any additions to $S$. The monotone operations $\cup$ and $\cap$ allow to directly propagate the expected results. Likewise, additions allow propagating increase and decrease requests.

Following the propagation of the expected result, potentially erroneous sub-expressions indicate where the expected result does not match the actual result. As a result of the applied grammar reflecting most model access operations, particularly

```
input: (e, v) ∈ {(Expression, Result)}, fCar
output: (h, m_s) ∈ {Car, Reqtable}
h ← {f_s|f_s ∈ f, ∃x : f_s = equal(t)}
m_s ← (a, b) ↦ {}
for equal(t) ∈ h
    s_r ← {(e_i, x_i)|(e_i, x_i) ∈ subresults(e, v))
    s ← generateChangeReq(e, x_1, ..., x_n, t)
    if s = ∅ //No fix, so don't handle!
        h = h \ equal(t)
    else
     for (e_i, x_i) ∈ s_r //Add requirement values
        m_s = (a, b) ↦ m_s ∪ {equal(t_i)|(t_1, ..., t_n) ∈ s}
```

Listing 4. Application of change action requirements generator operation for finite functions.

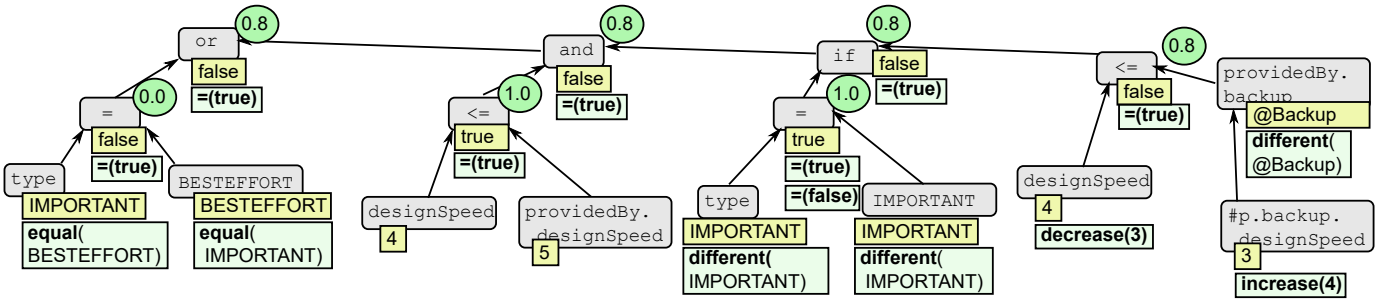


Fig. 4. Failed evaluation tree for the running example expression (cf. Listing 1).

object feature access operations and object instance selection operations, an error marker as well as annotations are placed at locations where sub-expressions are violated and therefore the expected result is not valid. By fulfilling change requirements for a sub-expression, this sub-expression could evaluate to true and thus contribute to making the whole expression true. It might be sufficient to fulfill a subset of all change requirements.

Let us consider the evaluation tree of the failed constraint of our running example as shown in Fig. 4. The syntactic tree is shown in a monospace font. Below each subexpression node, the current result is listed first, followed by the expected result. The numbers in the top right indicate the equality of actual result versus expected result. The constraint failed because the type was *IMPORTANT* and not *BESTEFFORT* and the design speed of the backup was not high enough. To find out what can be done to repair the constraint, we set the expected value to *true* and propagate it. The Boolean expression or is *true* if at least one subexpression is *true*, so we propagate the expected value *true* to both subexpressions. The first subexpression is an *equal* which is *true* if the value of *type* is equal to *BESTEFFORT*, so we set the expected value of *type* to *BESTEFFORT* and the expected value of *BESTEFFORT* to *IMPORTANT*. To make the and Boolean expression *true*, both subexpressions should return *true*. The first subexpression already returns *true* so there is nothing to propagate further and the *designSpeed* feature also does not need to be highlighted. However, the second subexpression does not yet return *true*. An if may evaluate to *true* if the condition is fulfilled and the then-subtree is *true* or the condition is not fulfilled and the else-subtree is *true*, so we try both cases in the conditional equals. This equals expression is currently *true* and can be made *false* by setting either subexpression to a different value. For space reasons, we only discuss making the then-part *true*. It is *true* if the *designSpeed* is at most the design speed of the backup, so the backup may be changed, the service's *designSpeed* may be decreased or the backup's *designSpeed* may be increased.

### C. Generic Approach for Advanced Content-Assist

Our approach ranks discovered content-assist suggestions, such that most favorable solutions are placed first. In principle, solutions violating the lowest number of constraints are considered favorable. Constraints are recursively evaluated by matching the closest match of a set of expected values with actual values of sub-expressions. The *closeness measure* in our approach is established by computing distance metrics for types. In detail, metrics for Strings and numbers are established by computing the Levenshtein distance and the numeric equality $e^{-|v_1-v_2|}$, respectively. Moreover, Boolean operators are mapped to double operators where *true* and *false* map to *0* and *1*, respectively. Additionally, and and max are mapped to min and max, respectively.

If we consider the example in Fig. 4, the only part of the expression which is not completely true or false is the rightmost inequality where the inequality is nearly fulfilled (measure 0.8). This value is propagated to the top so the constraint is considered mostly fulfilled.

### D. Generic Approach for Advanced Quick Fix Solutions

Quick fix solutions are actions—placed at locations where model values are changed—that can be executed to increase model quality by decreasing the number of violated constraints. Zeller defines the cause of an effect as the minimal difference between the world where the effect, i.e., a constraint violation in our case, occurs and an alternate world where the effect does not occur [28]. Hence, to find the actual cause for a constraint violation we have to search for the closest possible world in which the violation does not occur. Thus, we established quick fix solutions by ranking them according to the least change principle, i.e., favoring solutions causing minimal differences for how some effect comes to existence.

Our approach takes use of different SBSE-techniques for discovering quick fix solutions by applying a three-stage search process, which involves local as well as global search, with varying neighborhood search [29] for the exploration of the search space. Moreover, all involved search runs are both performed in the background and continuously deliver solutions that are eventually displayed to the editor user. In case a model change occurs, previously discovered quick fix solutions are re-evaluated based on the current model. In general, new values and value changes are randomly sampled for each specific data type. Moreover, changes are sampled in a way that small changes are more likely than large changes. For example, the likelihood of changing $n+1$ characters is only 10% of the likelihood of changing $n$ characters.

***Local Search.*** Our generic approach performs local search in two stages, i.e., by a small local search and a large local search. In general, model changes based on a single violated constraint are explored, which either resolve the violation,

i.e., constraint becomes true, or guide the constraint resolution closer to true, i.e., the involved model change has a high score of resolving the violation. A simple hill climbing algorithm with backtracking is used for the local search [30]. Of course, locally resolving an error may yield to new violations in other model areas. Therefore, as a countermeasure, our approach sorts quick fix solutions in terms of $(i)$ their score of resolving the involved single violation and $(ii)$ how many violations exist, i.e., involving existing violation and new violation, in the model after applying the solution.

The *small local search* explores model changes where the expression analyzer may find concrete possible changes for resolving a single expression. In detail, model changes include the application of equal, contains, excludes, increase, and decrease on objects features to reach the expected results. The attempted value of model change operations increase and decrease refers to the border of what is applicable.

The *large local search* considers changes for single violated expressions as well. However, it considers model changes involving manipulation of all feature values as well as all object instances that occur as part of a constraint expression in which the evaluated result does not match the expected result.

***Global Search.*** Our generic algorithm for global search applies genetic search techniques allowing arbitrary model changes on the entire model for discovering a broad range of quick fix solutions that may not yet have been discovered by local search algorithms. Hence, as a short-cut, quick fix solutions that result from the global search may directly be applied by the editor user. Practical implementations may consider multi-objective search algorithms, such as the NSGA-II algorithm, i.e., also applied in *IntellEdit*, which allows considering change amount as well as the number of fixed and violated constraints. Hence, the editor user is presented with the results of the current Pareto-front of such changes.

Our motivating example model (cf. upper-part of Fig. 1) may be repaired by a variety of repair actions that directly fulfill the expression and may thus be proposed to the editor user. For example, reducing the quality of the Service to *BESTEFFORT* and increasing the design speed of the backup.

### E. Running Advanced DSML Editors with IntellEdit

During the creation and manipulation of *DSML Model*s by the editor user (cf. Fig. 3), several interactions between the *Advanced DSML Editor* and *IntellEdit* occur to provide editor assistance. Moreover, to make the most appropriate use of resources consumed by our framework, any change made to the model is immediately communicated to our framework. The two major components of our framework are represented by the *IntellEdit OCL Interpreter* and the *IntellEdit Content-Assist and Quick Fix Builder*, which contains a *Local Search Engine* as well as a *Global Search Engine*.

The general workflow entails the delivery of identified error location details by our OCL interpreter to both our internal content-assist and quick fix builder as well as the external *Validation Provider* in the editor implementation. As a result, the editor user is provided with the validation results of the model being currently created or manipulated in the editor. Next, the editor's *Content Assist Provider* requests and retrieves any available results from our content-assist and quick fix builder. There, for our motivating example model, a line stating "`type = `" for our main Service will result in our content-assist builder to prefer to insert "`BESTEFFORT`" at the current cursor location. Finally, the editor's *Quick Fix Provider* is continuously being equipped with quick fix solutions by our content-assist and quick fix builder. Hence, as soon as solutions like changing the type to `BESTEFFORT` or increasing the Backup cluster speed to `4` are found, they are made available to the quick fix provider and can, therefore, be applied by the editor user operating the editor.

### F. Current Limitations

First, the propagation for operations with a finite operation table and a certain target value has (only) been implemented for the Boolean operations and, or, not, implies and xor. Secondly, if the type of a collection source set $S$ is finite, additions that do not yield a matching $f(y)$ are not (yet) avoided in our prototypical implementation. Third, the content-assist algorithm considers feature additions and updates from at most 1000 domain feature values. Forth, relying on the definition of cause and effect [28], applying IntellEdit quick fix solutions can make the failure disappear but may not necessarily represent a correction but at least a good workaround. Fifth, for specific expressions such as and, the OCL evaluation engine may only produce parts of the result, e.g., if the first subexpression is *false*, the second subexpression will not be evaluated which reduces the number of displayed validation errors. This could be fixed by changing the evaluation engine. Sixth, arbitrary function calls that combine the value of more than one variable can not be handled other than propagating change. Errors in an OCL constraint with such function calls may thus only be fixed by the second and third layer of our search algorithm.

## VI. Evaluation

The evaluation of our approach implemented by the IntellEdit framework has been conducted separately for our enhanced validation, content-assist suggestions, and quick fix solutions. We applied the DSML of our running example for randomized generations of example instances that each contains 20 objects as well as 100 associations and feature values.

### A. Validation

The validation results provided by IntellEdit have been evaluated based on whether it improves the estimation of locations where a model should be changed to resolve errors. We introduced erroneous changes at arbitrary locations starting from a valid model. We consider error feature locations to be accurate if they show exactly those locations as erroneous. In detail, we considered error locations to be correct if $(i)$ the feature is indicated as erroneous for every deleted feature value, $(ii)$ the container of the feature is indicated as erroneous for deleted objects, and $(iii)$ the whole object is indicated

TABLE II
VALIDATION EVALUATION RES. (WEIGHTED/UNWEIGHTED).

| Feature | IntellEdit | XText |
|---|---|---|
| **Precision** | 44.8% / 55.1% | 14.0% / 21.4% |
| **Recall** | 37.5% / 77.1% | 41.5% / 65.2% |
| **F-Measure** | 0.408 / 0.643 | 0.209 / 0.322 |

TABLE III
CONSTRAINT VIOLATIONS IN THE GENERATED MODELS.

| Use of IntellEdit Suggestions | 0% | 25% | 50% | 75% | 100% |
|---|---|---|---|---|---|
| **Avg. Cons. violations** | 8.8 | 6.5 | 3.8 | 2.4 | 0.0 |

as erroneous for created objects. Moreover, if a complete object is indicated as erroneous, all its features are indicated as erroneous as well. Finally, we determined precision and recall of our approach and of Xtext by comparing the set of erroneously indicated features and objects.

Table II shows the evaluation results of the IntellEdit validation compared with the Xtext validation for indicating erroneous locations in a model. In total, we performed 6442 evaluations on 50 generated models. The left-hand numbers represent the average precision and recall per single changed feature and the right-hand numbers represent the average precision and recall per single change. Hence, the left-hand numbers focus on large changes and the right-hand numbers on small changes. Our results state that the validation precision achieved by IntellEdit is nearly three times as high as that achieved by Xtext. Thus, the IntellEdit validation displays only erroneous features instead of indicating the whole object as erroneous and hence provides more accurate evidence on which parts of the model should be changed. In terms of recall, IntellEdit produces a higher recall per expression, i.e., performing better on expressions leading to small model changes, and a lower recall per edited feature. However, the recall for larger changes in our evaluation results is low in general. Hence, the validation of IntellEdit doubles the F-measure when compared with Xtext. To summarize, the validation results produced by IntellEdit improve existing precision three-fold while keeping recall measures of Xtext intact. Thus, IntellEdit indicates locations in the model that are more likely to be relevant for repairing violated constraints.

### B. Content-Assist Suggestions

The purpose of content-assist suggestions provided by IntellEdit is to support editor users in constructing consistent models. Hence, to evaluate content-assist suggestions, we first randomly generate model containment hierarchies and then assign features by using the result of our content-assist suggestions that has the highest score in 0% (i.e. completely randomly), 25%, 50%, 75%, and 100% of the cases, i.e., simulating the combination of random suggestions with IntellEdit-provided suggestions. Finally, we compare the number of violated constraints (cf. Table III).

Our results indicate that the IntellEdit content-assist provider is effective in preserving constraints in models. On one hand, highly scored IntellEdit content-assist suggestions, i.e., suggestions that are listed on top, do not introduce constraint violations in the model. On the other hand, valid random assignments that do not consider eventual restrictions introduced by formal constraints, i.e., similar to those suggested by the Xtext default content-assist provider, are likely to introduce constraint violations in the model.

### C. Quick Fix Solutions

The purpose of evaluating quick fix solutions provided by IntellEdit is to find out if they lead to a model with fewer constraint violations. In detail, we randomly generate models containing random values and apply up to ten IntellEdit quick fix suggestions created as a result of our three-stage search process. Moreover, to avoid the risk of simply deleting violated parts that subsequently may yield empty models, we do not select quick fix suggestions based on remaining constraint violations but on newly fulfilled violated constraints.

Our results show that applying quick fix suggestions on 97 randomly generated models leads to a reduction of the total number of violated constraints in 89 of the same models. Moreover, the number of violated constraints could not be improved in eight models. In detail, one model contained the same number of violations and seven models introduced further violations. To summarize, the quick fix solutions provided by IntellEdit were able to reduce the number of violated constraints from 7.9 to 2.6 (about 67%).

### D. Threats to Validity

In general, our evaluation is limited to our applied experimental object, i.e, the DSML for modeling service clusters, and hence may not be a representative of all kinds of languages. Moreover, the faults introduced in our faulty models may deviate from faults introduced by human users. Similarly, workflows of human users, which intend to resolve faults, may deviate from those suggested by our content assist and quick fix solutions. In fact, even for the running example, the typical change aimed at, i.e. increasing the speed of existing servers or adding new servers to meet the service requirements, is not found in our framework because making the contract weaker is considered as just a single change fulfilling all constraints while the change aimed for would need more model changes, resulting in high costs, without any intermediate improvements. Further, the error feature location in our evaluation assumes that a revert of the erroneous change is correct. However, there may exist different changes that can lead to good results. Hence, our evaluation does not clarify whether increased precision will also be beneficial for the editor user. Likewise, it is not known whether editor users mainly prefer to create models that do or do not violate constraints. For example, the content-assist provider of the Xtext default DSML editor allows to introduce violations that may or may not hinder the editor user in the creation of models. However, we hypothesize that picking content-assist suggestions that do not introduce violations, i.e., listed before other IntellEdit-provided content-assist suggestions, are more fruitful for the editor user. As a result, the current threats

TABLE IV
STRUCTURAL CONSTRAINTS IN LANGUAGE WORKBENCHES.

| Language Workbench | Formal Constraint Definition | OCL | Manual Constraint Validation | Automated Constraint Validation | Manual Scoping | Automated Scoping | Manual Quick Fix | Automated Quick Fix |
|---|---|---|---|---|---|---|---|---|
| Xtext | ◒ | ◒ | ● | ● | ● | ◒ | ● | ◒ |
| Jetbrains MPS | ◒ | ○ | ● | ● | ● | ● | ● | ○ |
| Rascal | ○ | ○ | ● | ○ | N/A[2] | ○ | ● | ○ |
| Melange | ◒ | ◒ | ● | ○ | ○ | ○ | ○ | ○ |
| Spoofax | ◒ | ○ | ● | ● | ● | ○ | ○ | ○ |
| Epsilon | ● | ● | ● | ○ | ○ | ○ | ● | ○ |
| Whole Platform | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| DrRacket | ● | ○ | ● | ● | ○ | ○ | ● | ○ |
| Eco | ○ | ○ | ● | ○ | ● | ○ | ○ | ○ |
| Ensō | ○ | ○ | ◒ | ◒ | ○ | ○ | ○ | ○ |
| MontiCore | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| MetaEdit+ | ● | ○ | ● | ● | ● | ○ | ● | ○ |
| SugarJ | ◒ | ○ | ● | ◒ | ● | ◒ | ○ | ○ |
| Visual Studio | ○ | ○ | ● | ○ | ● | ○ | ○ | ○ |

Legend: ● = Full Support; ◒ = Partial Support; ○ = No Support

to validity will be tackled in the future by conducting an industrial user study considering multiple real-world languages to evaluate whether our evaluation results can be reflected by an increase in usability of DSML editors as well as efficiency in the process of model creation carried out by human modelers.

## VII. STRUCTURAL CONSTRAINTS IN LANGUAGE WORKBENCHES

For the purpose of evaluating the applicability of our approach, which requires the specification of structural constraints as part of the language metamodel for creating automated support of advanced editing capabilities, we investigated existing languages workbenches listed by the Language Workbench Challenge 2016[3] that have been updated at least once within the last three years as well as Microsoft Visual Studio [31]. Hence, the intention of this comparison is to explore capabilities of existing language workbenches in terms of $(i)$ formal constraint definition, $(ii)$ manual specification and automated generation of constraint validation, $(iii)$ manual specification and automated generation of content-assist, and $(iv)$ manual specification and automated generation of quick fix solutions. Additionally, we intended to establish the compatibility of existing language workbenches with our approach and *IntellEdit* in particular.

[2]Not enough information for an evaluation of manual scoping in Rascal could be retrieved.

[3]Language workbenches listed by the Language Workbench Challenge 2016 can be found online at http://2016.splashcon.org/track/lwc2016.

In detail, we reviewed the following levels of language editor feature support offered by languages, which may be created by the language workbench under investigation. First, we examined the feasibility of language workbenches in their definition of languages that include formal constraints, such as OCL constraints. Second, we explored the range of support that is provided to language designers for manually specifying invariant-enforcement mechanisms in languages, which are generated by the investigated language workbench. Third, we inspected language workbenches in terms of their support for manually specifying content-assist amendment mechanisms, i.e., their intended capability for formulating content-assist suggestions, within applicable contexts. Fourth, we evaluated a language workbench's intended support for manual specification of quick fix solutions. Fifth, in the case of language workbenches that provide the capability of restricting languages by constructing formal constraints, we examined their degree of automating the generation of invariant-enforcement mechanisms as well as quick fix solutions available in resulting language editors.

Our review took into account several associated language workbench artifacts including $(i)$ documentation, $(ii)$ publications, $(iii)$ direct communication with developers of the language workbenches, and $(iv)$ implementation. We also took into account that a language workbench may not provide the support for structural constraints but still offer approaches for the implementation of customized validation, content-assist, and quick fix solutions. Note that, the "Onion" language workbench, listed in the Language Workbench Challenge, has been omitted because publication, documentation, and implementation is not publicly available. Hence, our investigation took into account Xtext [5], JetBrains MPS [32], Rascal [33], Melange [34], Spoofax [35], Epsilon [36], Whole Platform [37], DrRacket [38], Eco [39], Ensō [40], MontiCore [41], MetaEdit+ [42], SugarJ [43], and Microsoft Visual Studio [31]. Note that the results of our investigation (cf. Table IV), which includes direct communication with tool developers, may be still prone to misinterpretations. Hence, a more extensive comparison, e.g., involving the implementation of comprehensive DSMLs in each workbench, may produce deviating results.

We found that none of the workbenches offer both $(i)$ the capability of enhancing language definitions with formal constraints, such as OCL, and $(ii)$ the application of formal constraints in the automated generation of advanced editor support, i.e., validation, content-assist, and quick fix implementations. Moreover, we found that structural constraints imposed to languages, which are produced by the investigated language workbenches, is performed through either application of $(i)$ workbench-specific languages, $(ii)$ workbench-specific aspects, $(iii)$ GPL code, or $(iv)$ a combination of such.

Full support for structural constraints is provided in Epsilon, DrRacket, and MetaEdit+. However, Epsilon does not represent an actual language workbench, which may be used to generated a DSML, but a family of languages, including the Epsilon Validation Language (EVL), based on OCL. Hence,

EVL can be used to formulate constraints that may be used in conjunction with other tools, e.g., Xtext, in the creation of DSMLs. In detail, EVL may be used to augment and evaluate OCL invariants in Ecore models. Therefore, our approach can be applied in conjunction with EVL as well as profit from advantages of EVL over OCL [16], e.g., support for dependent constraints and enhanced flexibility in context definition. DrRacket allows formal specification as well as automated validation of contracts, manual validation by creating custom functions to display error locations, and specification of manual quick fix solutions by defining functions that make contracts valid. MetaEdit+ employs both a metamodeling language for the specification of graphs, objects, properties, ports, roles, and relationships as well as a scripting language for the specification and automated validation of structural constraints. Moreover, MetaEdit+ allows to manually specify validation, content-assist, and quick fix implementations. However, MetaEdit+ represents a non-generative approach, i.e., language definition and language use is done in the same tool instance, and hence only allows the non-generative part of our approach to be applied.

Xtext, JetBrains MPS, Melange, Spoofax, and SugarJ provide partial support for structural constraints. Both Xtext and Melange build on EMF by supporting language specifications in Ecore metamodels, which may be augmented with OCL constraints as done in our IntellEdit framework. Melange itself does not generate DSMLs but may be used in conjunction with other EMF-based tools, such as Xtext or Sirius [44], for generating DSML implementations. Hence, the capabilities of DSMLs specified with Melange are bound to EMF-based tools that are applied for generating language implementations. Additionally, Melange provides their own metalanguage for specifying model types based on the definition of groups of related types, i.e., a set of constraints over admissible model graphs. Both Xtext and Melange apply the default OCL interpreter provided by EMF, which offers automated validation that is limited in terms of displaying useful error messages. Moreover, Xtext also provides facilities for manually specifying content-assist and quick fix implementations and automatically offers basic content-assist and quick fix solutions for DSMLs.

The JetBrains MPS language workbench offers their own languages for specifying structural constraints. First, MPS offers automated validation and content-assist for constraints specified in their dedicated constraint language as well as manual specifications of validation, content-assist, and quick fix solutions by resorting back to GPL implementations. Second, structural constraints in MPS are limited to the expressiveness of their dedicated constraint language, which does not represent the full support by OCL. The Spoofax language workbench also employs their own constraint specification language which seems to only cover a subset of OCL. For example, Spoofax does not allow the specification of iterators in structural constraints. Hence, MPS and Spoofax may only partially profit from the advantages provided by our approach.

Restrictions in SugarJ may be defined using regular expressions that enable, together with imports of predefined syntactic extensions, validation, and content-assist capabilities. However, as a result of missing support for quick fix implementations as well as limited expressiveness of regular expressions, our approach may not be applicable in SugarJ.

Visual Studio offers no support for formal constraints. Validators and rules which change model elements depending on other model elements can be manually implemented to define validation, scoping, and some limited repairs. This approach could be applicable if OCL support would be added to Visual Studio, but this is unlikely since EMF/OCL is implemented in Java which is not the language of choice for Microsoft.

## VIII. Conclusion and Future Work

We presented an approach for leveraging language analysis for the automated generation of advanced editors in modern language workbenches. Hence, by applying our approach, language designers can construct formal constraints as part of their language definition to overcome the high effort of implementing advanced editing features by hand. Consequently, editor users are empowered with the benefits of such features, which include enhanced validation, content-assist suggestions, and quick fix solutions that are beyond those created by state-of-the-art language workbenches. The evaluation of our approach yields $(i)$ a two-fold improvement in validation over existing solutions, $(ii)$ content-assist suggestions that do not introduce new constraint violations, and $(iii)$ quick fix solutions that reduce the number of violated constraints by about $67\%$.

Concerning ongoing and future work, we are currently in the process of extending the evaluation of our approach by conducting experiments involving substantially larger DSMLs as well as DSMLs that are used in an industrial context. The results produced by such an evaluation will provide further insights on performance, scalability, and practical feasibility of our approach. Moreover, we intend to precisely determine any advances that the application of our approach may introduce. In detail, an extensive comparison of IntellEdit with other solutions will be conducted by implementing a comparable DSML in each of them and evaluate their capability in terms of provided validation, content assist, and quick fix solutions. Further, we want to measure the effect of applying different primitive type conversions, e.g., multiplication instead of min for the and operator, on the quality of produced content assist and quick fix results. Finally, in case additional evaluation results provide further evidence in favor of applying our approach, we plan to perform an empirical study, which will be conducted within an industrial context, measuring the effects on efficiency and productivity of language designers as well as editor users applying our approach.

## Acknowledgment

We graciously thank the authors and developers of the language workbenches mentioned in Section VII for their effort in supporting us by discussing the capabilities of their tools.

## REFERENCES


[1] S. Kelly and J.-P. Tolvanen, *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[2] J. Gray, S. Neema, J. Tolvanen, A. S. Gokhale, S. Kelly, and J. Sprinkle, "Domain-Specific Modeling," in *Handbook of Dynamic System Modeling.*, 2007.

[3] U. Frank, "Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges," *Software and System Modeling*, vol. 13, no. 3, pp. 941–962, 2014.

[4] M. Fowler, "Language workbenches: The killer-app for domain specific languages," 2005. [Online]. Available: http://www.martinfowler.com/articles/languageWorkbench.html

[5] M. Eysholdt and H. Behrens, "Xtext: Implement your Language Faster than the Quick and Dirty Way," in *Companion Proc. of OOPSLA*. ACM, 2010, pp. 307–309.

[6] J. Favre, "Languages evolve too! Changing the Software Time Scale," in *Proceedings of IWPSE*, 2005, pp. 33–44.

[7] B. Meyers and H. Vangheluwe, "A framework for evolution of modelling languages," *Sci. Comput. Program.*, vol. 76, no. 12, pp. 1223–1246, 2011.

[8] J. Cabot and M. Gogolla, "Object constraint language (OCL): A definitive guide," in *Proceedings of SFM*, 2012, pp. 58–90.

[9] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," in *Proceedings of ICSE*, 2003, pp. 455–464.

[10] T. Mens, R. V. D. Straeten, and M. D'Hondt, "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis," in *Proceeding of MODELS*, 2006, pp. 200–214.

[11] P. Neubauer, "Towards model-driven software language modernization," in *Proceedings of the Doctoral Symposium and Projects Showcase @ STAF*, 2016, pp. 11–20.

[12] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[13] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[14] J. Whittle, J. E. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.

[15] P. Neubauer, A. Bergmayr, T. Mayerhofer, J. Troya, and M. Wimmer, "XMLText: From XML Schema to Xtext," in *Proceedings of SLE*, 2015, pp. 71–76.

[16] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages," in *Rigorous Methods for Software Construction and Analysis*, 2009, pp. 204–218.

[17] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *Proceedings of ASE*, 2008, pp. 99–108.

[18] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *Proceedings of ASE*, 2012, pp. 220–229.

[19] ——, "Determining the cause of a design model inconsistency," *IEEE Trans. Software Eng.*, vol. 39, no. 11, pp. 1531–1548, 2013.

[20] M. Harman, "The Current State and Future of Search Based Software Engineering," in *Proceedings of FOSE*, 2007, pp. 342–357.

[21] Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró, "Quick fix generation for dsmls," in *Proceedings of VL/HCC*, 2011, pp. 17–24.

[22] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards Automated Inconsistency Handling in Design Models," in *Proceedings of CAiSE*, 2010, pp. 348–362.

[23] S. Ali, M. Z. Z. Iqbal, A. Arcuri, and L. C. Briand, "A Search-Based OCL Constraint Solver for Model-Based Test Data Generation," in *Proceedings of QSIC*, 2011, pp. 41–50.

[24] O. Semeráth, Á. Barta, Á. Horváth, Z. Szatmári, and D. Varró, "Formal validation of domain-specific languages with derived features and well-formedness constraints," *Software and Systems Modeling*, pp. 1–36, 2015.

[25] K. Deb, "Multi-objective optimization," in *Search methodologies*. Springer, 2014, pp. 403–449.

[26] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[27] "Multiobjective Evolutionary Algorithms (MOEA) framework," http://moeaframework.org, accessed: 2016-12-15.

[28] A. Zeller, *Why programs fail - a guide to systematic debugging*. Elsevier, 2006.

[29] P. Hansen and N. Mladenović, "Variable neighborhood search," in *Search methodologies*. Springer, 2014, pp. 313–337.

[30] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*. Prentice Hall, 2003, vol. 2.

[31] "Modeling SDK for Visual Studio," https://msdn.microsoft.com/en-us/library/bb126413.aspx, accessed: 2016-12-15.

[32] M. Voelter, "Language and IDE modularization and composition with MPS," in *Proceedings of GTTSE*, 2011, pp. 383–430.

[33] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM*, 2009, pp. 168–177.

[34] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J. Jézéquel, "Melange: a meta-language for modular and reusable development of DSLs," in *Proceedings of SLE*, 2015, pp. 25–36.

[35] L. C. L. Kats and E. Visser, "The spoofax language workbench: rules for declarative specification of languages and IDEs," in *Proceedings of OOPSLA*, 2010, pp. 444–463.

[36] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack, "The Epsilon Generation Language," in *Proceedings of ECMDA-FA*, 2008, pp. 1–16.

[37] R. Solmi, "Whole platform," Ph.D. dissertation, University of Bologna, 2005.

[38] M. Flatt, "Creating languages in Racket," *Communications of ACM*, vol. 55, no. 1, pp. 48–56, 2012.

[39] L. Diekmann and L. Tratt, "Eco: A Language Composition Editor," in *Proceedings of SLE*, 2014, pp. 82–101.

[40] T. van der Storm, W. R. Cook, and A. Loh, "Object Grammars," in *Proceedings of SLE*, 2012, pp. 4–23.

[41] H. Krahn, B. Rumpe, and S. Völkel, "Monticore: a framework for compositional development of domain specific languages," *CoRR*, vol. abs/1409.2367, 2014.

[42] K. Smolander, K. Lyytinen, V. Tahvanainen, and P. Marttiin, "MetaEdit - A Flexible Graphical Environment for Methodology Modelling," in *Proceedings of CAiSE*, 1991, pp. 168–193.

[43] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, "SugarJ: library-based syntactic language extensibility," in *Proceedings of OOPSLA*, 2011, pp. 391–406.

[44] V. Viyovi, M. Maksimovi, and B. Perisi, "Sirius: A rapid development of DSM graphical editor," in *Proceedings of INES*, 2014, pp. 233–238.