Robust Projectional Editing

Friedrich Steimann
Fernuniversität in Hagen
steimann@acm.org

Marcus Frenkel
Fernuniversität in Hagen
marcus.frenkel@feu.de

Markus Völter
independent/itemis AG
voelter@acm.org

Abstract

While contemporary projectional editors make sure that the edited programs conform to the programming language's metamodel, they do not enforce that they are also well-formed, that is, that they obey the well-formedness rules defined for the language. We show how, based on a constraint-based capture of well-formedness, projectional editors can be empowered to enforce well-formedness in much the same way they enforce conformance with the metamodel. The resulting *robust edits* may be more complex than ordinary, well-formedness breaking edits, and hence may require more user involvement; yet, maintaining well-formedness at all times ensures that necessary corrections of a program are linked to the edit that necessitated them, and that the projectional editor's services are never compromised by inconsistent programs. Robust projectional editing is not a straitjacket, however: If a programmer prefers to work without it, its constraint-based capture of well-formedness will still catch all introduced errors — unlike many other editor services, well-formedness checking and robust editing are based on the same implementation, and are hence guaranteed to behave consistently.

CCS Concepts **•Software and its engineering → Software maintenance tools; Semantics; Syntax; •Theory of computation → Constraint and logic programming;**

Keywords projectional editing, constraint propagation, editors, well-formedness, static semantics

1 Introduction

> "robust, *adj.* 1. strong and healthy"
> [Webster's Encyclopedic Unabridged Dictionary of the English Language]

Projectional editing means editing the internal representation of a program via projection to one or more surface notations (which may, but need not be, textual) [36]. For textual languages, it does not rely on a standard text processor, but rather on a structure editor based on templates which are defined as part of the language specification, and which the user needs to fill in. While still somewhat controversial among programmers (see Section 8 for a discussion), projectional editors excel at supporting them with entering valid programs, by exploiting available structural and contextual information in the editing process [1, 14, 38].

However, even with projectional editors, it is easy to enter malformed, or to break well-formed, programs. For instance, changing the declared type of an entity may invalidate uses of this entity. Malformedness is a problem for editor services that depend on program analyses that in turn depend on well-formedness of programs (see [14] for a recent account), such as code completion and refactoring tools (which may produce incomplete or incorrect results if applied to malformed programs). On a larger scale, malformedness is also a problem for collaboration, namely if commits are not allowed to break programs, and are required to be frequent and small (to increase parallelism and to avoid merge conflicts).

Robust editing as defined in this paper means *editing that leaves a well-formed program well-formed* (and that is hence healthy and strong). As we will see, robust editing may require individual, intended edits to be complemented by additional, complementing edits that maintain (or restore) well-formedness; yet, robust edits are atomic in the sense that they are either performed completely or not at all.

We present an implementation of robust editing as a new language service of the projectional editing environment MPS[1]. Our *robust projectional editor* (RPE) builds on two DSLs: one for specifying well-formedness, and one for specifying robust editing patterns (named *edit intentions*). The former replaces MPS's currently different DSLs for type and other checking rules, as well as its (again different) DSL for scope constraints, not only providing for a single point of reference for a language's static semantics, but also allowing all rules to be exploited in concert for computing robust edits. The latter provides for the specification of a catalog of named editing patterns that are seamlessly integrated with existing language services of MPS.

While our implementation of robust editing draws on insights gained from our works on constraint-based refactoring and repair [22, 26–29], it differs substantially not only in its purpose, but also by replacing standard constraint solving with custom implementations of constraint propagation. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE'17, Vancouver, BC, Canada
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5525-4/17/10...$15.00
DOI: 10.1145/3136014.3136034

[1] https://www.jetbrains.com/mps/

latter allows us to introduce special purpose constraints and, more importantly, to interleave the search for a solution with user interaction, thus avoiding the computation of solutions that the user will dismiss. Our work not only fills a hole in the array of language services that can be generated from a single constraint-based specification of a language's static semantics [25], it also responds to a recent call for researching the semantic foundations of editors [14].

Our paper is organized as follows: After a motivating example (Sect. 2), we lay the foundations of robust projectional editing (Sect. 3) and state the problem we are addressing (Sect. 4). We present a generic solution of the problem (Sect. 5) and complement it with edit intentions (Sect. 6). After a detailed account of our implementation (Sect. 7), we discuss our work (Sect. 8) and relate it to others (Sect. 9).

2 Motivating Example

To motivate our notion of robust edits, we assume here a simple language with modules and records (LMR; see [33] for its definition, and Figure 1 for an excerpt of its abstract syntax), in which we have written the (incomplete) program

```
prog
  module A record Y { c : X } end
  module B record Y { a : X } end
  import A
  record X { a : X b : X }
  def x : X = new X {}
  def y : Y = new Y {}
  def z = x._
end
```

At the position of the underscore (the current editing position), the RPE that we envision would offer us a choice of {a, b} for selection. The RPE can infer this choice from (i) structural knowledge, knowing that, in LMR, only a reference to a record field can occur on the right of the dot, restricting the choice to {a, b, c, B.Y.a}; and from (ii) program-level type information, knowing that the type on the left of the dot, the type of x, is X, which has members a and b, thus restricting the choice to {a, b}. Note that, were the field access a entered before the receiver x, the choice for the field would be {a, b, c} (using scope information to exclude B.Y.a); after selecting a, well-typedness would require the receiver (left of the dot) to have type X, including x, but ruling out y for selection.

Suppose that we have entered x.a, but now want to replace x with y. As noted above, this would render the program ill-typed. However, rather than preventing this editing intent, a more useful RPE would allow the user to choose y anyhow, requiring selection among a number of options that maintain the program's well-formedness. Option ① would be to also replace a by c; option ② would be to move the field declaration a from type X to type Y, option ③ would be to change the type of y and its initializing expression to X; and option ④ to import module B instead of A, replace the type A.Y of y with B.Y, and select B.Y.a instead of X.a[2]. Last but not least, as option ⑤ the field access can be removed. Note that each option results in a well-formed program, in which (with the exception of the last) x in x.a has been replaced by y, the original editing intent; the last one is somewhat special in that it deletes a program element (rather than changing an existing one).

If deleting is an edit option, then adding should also be. Indeed, there is yet another option ⑥ that users of a standard editor can readily choose, by going through a temporarily malformed program: replace x with y and add and select a new field a to A.Y. In fact, adding new program elements only after using them in a program under edit is a common programming practice, one that contemporary projectional editors do not support, because every reference eagerly binds to its target when it is entered. To automatically complement a reference to a new program element with creating the element, an RPE would need to infer, from the current edit position, where the new program element (here: a new field a) is to be added (here: to A.Y).

3 Foundations

We define here the context in which we assume our RPE to operate: the abstract syntax, or language model, of the object language (i.e., the software language the RPE is provided for), and the object language's rules of well-formedness (static semantics; but recall that for projectional editing, this usually excludes rules of name binding; cf. Footnote 2).

3.1 Language Model

Following Lämmel [7], we assume that a program is represented by an attributed abstract syntax graph (ASG), where each node is an instance of a *struct*[3]. The edges of the ASG are encoded using reference attributes of the nodes, which are complemented by attributes holding values.[4] It follows immediately that the *primitive edit operations* we need to consider are

- *adding and deleting nodes of an ASG*, and
- *assigning values or references to attributes.*

Note that any ASG can be built (from scratch) using these operations, and can be changed in every conceivable way. Thus, the primitive edit operations suffice to describe all changes a programmer may wish to make to a program[5].

Further following Lämmel [7], we assume a conformance relation of ASGs to structural constraints expressed by a

[2]Projection implies that references are not re-computed by resolving names – they must be updated manually. See Section 8 for a discussion.
[3]We us the term *struct* rather than *type* here in order to avoid confusion with the types defined and used by programs. *Classifier*, *class*, *sort*, *syntactic category*, or *concept* are also in use; we prefer *struct*, because it suggests the notion of well-structuredness which we will introduce below.
[4]In OMG terms, our attributes would be called *properties*, which comprise value attributes (called *attributes*) and reference attributes, called *links*.
[5]From here on, we use the terms *ASG* and *program* interchangeably.

metamodel of the object language, such as that provided in Figure 1 (for the language LMR [33], adopted here for its focus on scope). Specifically, for an ASG to conform to a metamodel we require that

- each node of an ASG is an instance of a struct declared by the metamodel,
- the attributes attached to a node are precisely those declared as members of its struct or inherited from its superstruct (declared in Figure 1 using "`<:`"),
- the values or references held by attributes are elements of the attributes' declared value types or references to instances of the attributes' declared structs or superstructs, resp., taking the attributes' declared multiplicities into account, and
- each reference held by an attribute references an existing node (i.e., there are no dangling references).

Here, value (or data) types, which are not defined by the metamodel, are `str`, `int`, `bool`, etc.; multiplicities are either *one* (the default), or *option* (declared by appending "?" to the attribute's declaration; cf. Figure 1), or *many* (appending "`*`"). We call an ASG that conforms to a metamodel *well-structured.*

Once more following Lämmel [7], we assume that an ASG is overlaid by a tree structure, which is established by a bidirectional *parent-child* (or *containment*) relationship and which is implemented by using special reference attributes (declared with the keyword `child` in place of `ref` in the metamodel of Figure 1; the reverse direction is established by an implicit `parent` attribute). While such a tree structure may not be needed, in practice, we find that most software languages suggest it, if only to define a linear representation of the program, but usually also to express a *whole-part* relationship (constituency) and hierarchically nested scopes. If it is used by a metamodel, well-structuredness encompasses that the parent-child relation is a tree (or a forest).

While well-structuredness is usually considered a prerequisite for deciding well-formedness (see Section 3.2), primitive edit operations may lead to states in which the ASG is not well-structured. Specifically, after adding a new node, mandatory attributes (with multiplicity *one*) may not have been assigned yet[6], and after deleting a node, references may be dangling. We call an ASG that is well-structured except for possibly unassigned attributes or dangling references, *pre-structured*, and note that pre-structured ASGs can be made well-structured by assigning values or references to attributes; i.e., by primitive edit operations.

Finally, we assume that a projectional editor always leaves a pre-structured ASG at least pre-structured, and furthermore that the only way a projectional editor can make a well-structured ASG pre-structured is by adding or deleting nodes. It follows that if a program is entered entirely using a projectional editor, it is pre-structured at all times. To ensure that no ill-structured ASGs result from assigning reference attributes, a projectional editor will offer only the existing nodes (instances) of the attribute's declared struct or substructs for selection (so that no dangling references or references to nodes of other structs can be assigned). Note that some projectional editors (including MPS) may further restrict the selection of references by taking scope rules into account, but we will not treat scope rules differently from other rules of well-formedness, and hence do not consider this possibility at this point.

[6]Note that this does not mean that the attributes are assigned `null`; rather, their dynamic multiplicity [24] is — illegally — zero.

```
struct Prog {child decls:Decl*}
abstract struct Decl {}
struct Module <: Decl {
  val id:str;
  child decls:Decl*
}
struct Import <: Decl {ref module:Module}
struct Def <: Decl {
  val id:str;
  ref type:Record;
  child exp:Exp
}
struct Record <: Decl {
  val id:str;
  child fields:Fdecl*
}
struct Fdecl {val id:str; ref type:Record}
abstract struct Exp {ref type:Record}
struct New <: Exp {
  ref record:Record;
  child inits:Fbind*
}
struct DotExp <: Exp {
  child left:Exp;
  ref field:FDecl
}
struct VarRef <: Exp {ref var:Def}
struct FBind {
  ref field:Fdecl;
  child exp:Exp
}
```

Figure 1. Specification of graph-based abstract syntax (metamodel) of LMR (simplified excerpt).

3.2 Well-formedness

Most object language specifications impose constraints on ASGs that go beyond well-structuredness. Notably, this includes the rules of the used type system and the scope rules of a language. Although these rules are traditionally handled separately, we do not do so and instead express all rules

```
context Def inv "var type":
  self.type = self.exp.type
context Def inv "unique var names"
  group d1, d2 by parent:
    d1.id ≠ d2.id
context Def inv "type access":
  self.type.parent = self.parent.parent* or
    self.type.parent in
      self.parent.decls<Import>.module
context DotExp inv "field access":
  self.field.parent = self.left.type
context Fdecl inv "unique field names"
  group f1, f2 by parent:
    f1.id ≠ f2.id
context VarRef inv "var ref type":
  self.type = self.var.type
context DotExp inv "dot type":
  self.type = self.field.type
```

Figure 2. Some invariants for LMR expressed in our constraint DSL used by our RPE to check well-formedness and to infer types.

of well-formedness jointly, using the same constraint language.[7] Note that to be able to evaluate well-formedness rules on an ASG, both the ASG and the rules must conform to the metamodel (because otherwise, a rule might attempt to access an attribute that is not declared for a node, so that evaluation gets stuck); while evaluation of well-formedness rules usually requires that ASGs are well-structured, we will relax this condition in Section 5, requiring only their pre-structuredness.

Figure 2 shows type, scope, and other well-formedness rules for LMR, written in a constraint language designed for this purpose. Following the example of the Object Constraint Language (OCL) [13], each rule is attached to a struct of the metamodel, which provides the contexts of its evaluation. For instance, the rule "var type" is associated with the struct `Def` (see Figure 1), and is applied to each of its instances (represented in its body by the special variable `self`). The `group ...` by clause used in "unique var names" introduces two variables `d1` and `d2` bound to all instances of the context `Def` so that `d1` ≠ `d2` and both share the same parent.

As in OCL, reference attributes are implicitly dereferenced by the dot operator. For instance `self.exp.type` (from "var type") refers to the `type` attribute associated with the node referred to by `self.exp`. This allows us to express scope rules as *path expressions*. As a specialty, we introduce the star ("*") for navigating an attribute transitively zero to *n* times: for instance, `parent*` in `self.type.parent = self.parent.parent*` (from "type access") means that the node denoted by the left-hand side must be reachable by transitively navigating the `parent` attribute, starting from `self.parent`. This saves us from having to pass sets of declared entities "in scope" (symbol tables, or environments) around.

Also following OCL, path expressions can be many-valued. For instance, `self.parent.decls<Import>.module` (from rule "type access") denotes many references to modules, even though each attribute `module` holds only a single reference (the operator `in` tests membership as usual). Here, the expression `decls<Import>` selects all import declarations from the declarations held by `decls`. Note that while these special constructs of our constraint DSL are straightforward to evaluate (as required for checking well-formedness), they are a challenge for constraint propagation, as we require in the implementation of our RPE.

It is important to note that while the rules of Figure 2 define well-formedness for a well-structured ASG, some of them can also be used to infer values or references for attributes. Notably, "var ref type" and "dot type" serve to compute the type of expressions, and "var type" can be used to infer the type of a variable not specified in a `def` (e.g., `z` in our example). This will be returned to in Section 5.

4 Problem Statement

While we may assume that projectional editors always produce ASGs that are at least pre-structured (see Section 3.1), we may not expect guarantees with respect to the well-formedness of the resulting ASG (as noted in Section 3.2, for an ASG that is not well-structured, well-formedness may not even be defined). Hence, *the problem of robust projectional editing* is to complement the primitive edit operations offered by a projectional editor, which may make the ASG malformed or its well-formedness undefined, with the primitive *complementing* edit operations that are required to obtain a well-formed ASG. *The problem of implementing an RPE* is thus (1) to compute the complementing edit operations required for each offered primitive editing operation, and (2) to replace the offered primitive editing operations with (complemented) tuples of primitive edit operations — called *robust edit operations* — from which the user can select. The execution of a selected robust edit operation must be indivisible: either all comprised primitive edits are executed together, or none at all.

Leaving questions of user interface (UI) aside, the computation of the complementing edit operations is complicated by a number of technical challenges:

- For the complementing edit operations there may be many alternatives, none of which can be excluded a priori. Generally, computing complementing edits is a search problem, which may be difficult to confine.

[7] That scope rules and type rules may be difficult to separate can be seen by the static imports of Java, and also how in Java the availability of type members depends not only on the type of the receiver, but also (via access modifiers) on the type's and the receiver's locations in the program.

- If reference attributes are dereferenced in the course of checking well-formedness (cf. Section 3.2), complementing edits may affect dereferencing (by reassigning reference attributes). This complicates the search considerably.
- The search is further complicated by the *-operator (introduced in Section 3.2), which may involve an arbitrary number of reference attributes, all of which may be re-assigned.

The following two sections deal with the specifics of our solution on a more abstract level; Section 7 presents details of the actual implementation.

5 Generic Solution

The guiding idea of our solution is to encode well-formedness as a *constraint satisfaction problem* (CSP) [31], and to compute robust edit operations using constraint solving techniques. As usual, a CSP is defined here as a set of *constraint variables*, each associated with a *domain* from which its possible values are drawn, and a set of *constraints*, or relations, on these variables. For the encoding, we map

1. the attributes of an ASG to constraint variables,
2. the extensions of the structs and value types of an ASG to domains of the variables, and
3. the well-formedness rules to constraints on the variables.

Note that the second leg of the mapping includes a mapping of the nodes of an ASG to values of domains of the CSP, and of values and references held by the nodes' attributes to values of the corresponding constraint variables. Also note that for the mapping to be well-defined it suffices that the ASG is pre-structured (see Section 3.1 for the definition); unassigned attributes or reference attributes holding dangling references are mapped to unassigned constraint variables.

To illustrate the mapping, we revisit the (incomplete) definition `def z = x._` from our motivating example, which is represented by the ASG nodes

$$[\texttt{id} = \text{‘z’}, \texttt{type} = _, \mathit{exp} = \uparrow 1]_0 : \texttt{Def}$$
$$[\texttt{left} = \uparrow 2, \texttt{field} = _, \texttt{type} = _]_1 : \texttt{DotExp}$$
$$[\texttt{var} = \uparrow 3, \texttt{type} = \uparrow 4]_2 : \texttt{VarRef}$$
$$[\texttt{id} = \text{‘x’}, \texttt{type} = \uparrow 4, \ldots]_3 : \texttt{Def}$$
$$[\texttt{id} = \text{‘X’}, \ldots]_4 : \texttt{Record}$$

Here, an integer subscript i marks the target of all references $\uparrow i$, a node that is an instance of the struct given after the colon; the underscore marks an attribute whose value or reference has not yet been set. Each attribute a of each node i is then mapped to a constraint variable $\uparrow i.a$, whose initial value is the value or reference provided in the ASG. Invariants like those of Figure 2 are mapped to constraints by applying them to the instances of their context structs; for instance, applying the invariants "var type" and "dot type" from Figure 2 to the above instances (nodes of the ASG) produces the constraints

$$\uparrow 0.\texttt{type} = \uparrow 0.\texttt{exp.type}$$
$$\uparrow 3.\texttt{type} = \uparrow 3.\texttt{exp.type}$$
$$\uparrow 1.\texttt{type} = \uparrow 1.\texttt{field.type}$$

Note that, once the `field` attribute of node 1 has been assigned, the reference of the `type` attribute of node 0 can be computed by constraint solving, and that the same constraints can be used to check the well-typedness of a definition in which the type of `z` is declared (rather than inferred, as in our example). This supports our claim that we make do with a single specification of static semantics for all purposes.

A more formal account of how invariants of an OCL-like language can be mapped to constraints amenable to constraint solving can be found in our recent publication on partially evaluating OCL expressions [32]; in this present work, we deviate from this approach in that we do not actually generate the constraints (and do not use a constraint solver for solving them), but rather propagate them on the fly (see Section 7).

Our mapping of an ASG and the well-formedness rules of the language to a CSP is meaning-preserving in the sense that if and only if the ASG is well-formed, the CSP is satisfied with variable assignments representing the ASG. This means in particular that the variable assignment representing a well-formed ASG before a robust edit, and the variable assignment representing the ASG after it, satisfy the CSP. Also, every changed variable assignment of the CSP represents an edited ASG (and with it an edited program), namely one in which attributes (in case of reference attributes: edges!) have changed. Note that assigning values to unassigned constraint variables corresponds to making a pre-structured ASG well-structured, meaning that we can use our mapping to make a pre-structured ASG (for which well-formedness is undefined; see Section 3) resulting from a primitive edit well-formed.

While the mapping of nodes and their attributes is straightforward, the mapping of well-formedness rules is complicated by implicit and explicit quantification over the extension of structs which, since CSPs do not cater for quantification, must be unrolled. In particular, for well-formedness rules relying on the dereferencing of reference attributes, we cannot assume that the references do not change (reference attributes are also mapped to variables, which may be reassigned by constraint solving), so that dereferencing must be encoded explicitly in the constraints the well-formedness rules are mapped to. For instance, the constraint variable represented by $\uparrow$`1.field.type` changes with the value of $\uparrow$`1.field`, so that the mapping of well-formedness rules containing this expression to constraints must unroll over the extension of $\uparrow$`1.field`'s declared struct (`Fdecl` in our example). While there are various ways to accomplish this (see Section 7 for our current solution, and Section 9 for other

approaches we are aware of), unrolling remains a problem when the extensions (domains) being unrolled over are themselves variable, which is inherently the case when members are added or removed as part of editing (see below).

5.1 Computing Robust Edits

Given the above mapping from ASGs to CSPs, the primitive edit operations of Section 3.1 (recall that these are all we need to consider) correspond to changes of CSPs as follows:

- Assigning a value or a reference to an attribute corresponds to assigning the mapped value or reference to the mapped constraint variable.
- Adding a node to an ASG corresponds to adding (a) the mapped value to those domains of constraint variables that are mapped from the struct the node is an instance of, (b) the constraint variables corresponding to the attributes of the node, and (c) the constraints that constrain these variables.
- Deleting a node corresponds to deleting (a) the mapped value from the domains determined as above, (b) the constraint variables, and (c) the constraints.

Note that for adding, the CSP must fully cover the new node, as demanded by items (a) through (c) above. (c) is complicated by the fact that the added node has not been considered in the unrolling of any of the quantifications mandated by mapping the rules of well-formedness for the original CSP, including those resulting from path expressions (see above). There are two ways to ensure this: (1) re-map the ASG to a CSP after the node has been added, and (2) assume a reservoir of unused nodes that become used by referencing them. The former will preclude constraint solving from computing add operations as complements of a given refactoring intent, and hence limit the power of an RPE; the latter requires that the unused nodes are "parked" so that they do not appear in projections of the ASG. For deleting, the situation is analogous: the ASG is either remapped, or the deleted nodes are parked, making sure that they are no longer referenced. Note that the latter enables a wide range of complementing edits: for instance, when deleting a parent node, its children (who lost their parent) can either be deleted (not referenced) as well, or be assigned another parent (including the parent of their original parent, if this is covered by the declared parent-child relationship; we will present an example in Section 6).

To summarize, we have that by expressing an editing intent, a user identifies one or more attributes that are to be assigned. To make sure that the pending assignments make the ASG well-structured and well-formed, we map the ASG to a CSP and subject this CSP to constraint solving. Each solution of the CSP is comprised of specific variables assignments and hence presents an alternative robust edit operation, from which the user can select to fulfill the editing intent.

Simple as this approach may seem, it has to master two nontrivial practical problems:

1. The CSP encoding an ASG and its well-formedness is too big to handle in the general case.
2. The CSP may have too many solutions to choose from.

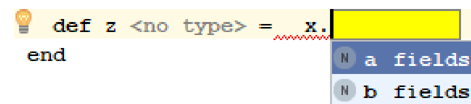
The problems are related in that if we find ways to manage that the CSPs become smaller, we will also receive fewer solutions. However, since "fewer solutions" does not automatically converge to "the solutions that the user appreciates", we must involve the user in exploring the solution space.

5.2 Exploring the Solution Space

As suggested above, exploring the solution space is comprised of controlling its size and selecting a solution.

5.2.1 Controlling the Solution Space

Without further input from the user, the variables selected by the editing intent are the only ones we allow the constraint solver to re-assign, with three possible outcomes: the corresponding CSP is unsatisfiable, its solutions appear unsatisfactory to the user, or the user selects one for execution. For instance, in the context of our motivating example of Section 2, our RPE offers a choice of `a` and `b`

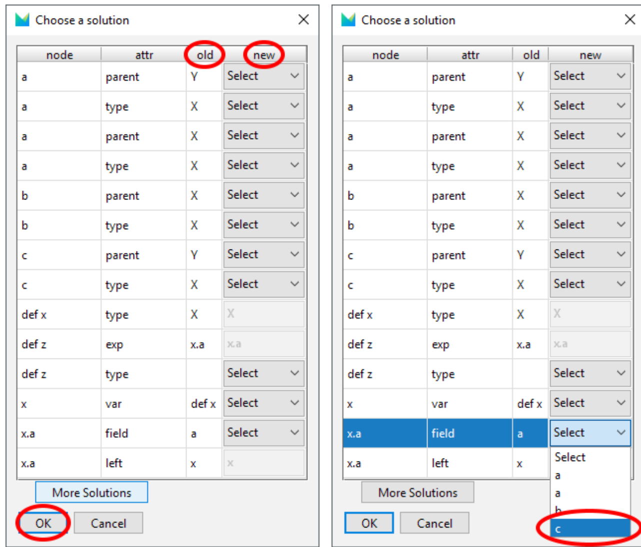


selecting from which assigns the corresponding reference to the constraint variable representing the reference attribute `field` of the instance of struct `DotExp` representing the (incomplete) dot expression, satisfying the constraint "field access" (the only constraint of Figure 2 involving `field` variables).[8] If this choice appears unsatisfactory to the user, more solutions (which would also be required by an unsatisfiable CSP) are obtained by either adding program elements, or by making more attributes re-assignable; specifically by making those attributes assignable that constrain the currently assignable attributes directly (in the above example, by adding a new field to record `X`, by making `type` of the receiver held by `left` assignable, or by making `left` itself assignable). This process can be repeated until a (satisfactory) solution is found. Ultimately, however, it will lead to a new problem: the problem of an exploded solution space, which the user has to conquer.

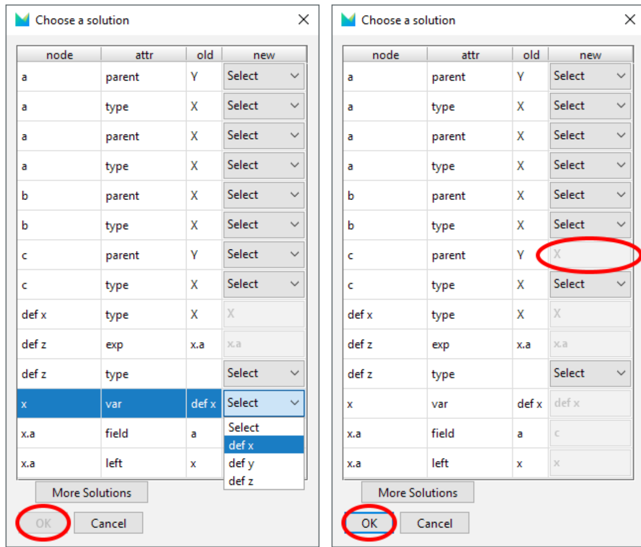
5.2.2 Selecting a Robust Edit

Selecting from a large solution space may be challenging (if only for the many repeating groups that alternative solutions will contain) and in any case is rather remote from projectional editing as we know it. We therefore project the set of alternative robust edits to the attributes involved, so that for each attribute, the user gets a choice of values or references. The choices are dependent in that every selection

[8] Incidentally, these are the same choices that would be offered by a standard implementation of LMR in MPS, but only if the type rules are implemented as scope rules; see Section 9.



(a) Looking at the solution space. **(b)** Selecting field c.



(c) Confirming variable x. **(d)** Confirming that c has moved from Y to X.

Figure 3. Robust edition session (all screenshots taken from our implementation described in Section 7).

of a value or reference for one of these attributes restricts the available choices for the remaining attributes, until no more choices remain, which is when a robust edit has been selected. The individual choices, of which there will be at most as many as attributes are involved, must be enclosed in transaction braces (*begin edit* and *end edit*) so as to guarantee that well-formedness is accomplished (or the editing intent given up) before the next robust edit commences.

To see this working, we have adopted our Solution Space Explorer [26] as a prototype UI for robust projectional editing. Continuing the motivating example where we left in Section 5.2.1, and assuming that the user found her selection of a unsatisfactory (and therefore asked for more alternatives), our RPE responds with the dialog shown in Figure 3a. It shows all attributes that immediately (i.e., through no more than one constraint) constrain the unsatisfying choice, and offers alternative assignments for some (column "new"). That the current assignment ("old") makes the program well-formed is indicated by the fact that the "OK" button is active.

From the offered alternatives, the user can make selections in arbitrary order. For instance, if c is the field that is to be accessed on x, this can be selected first (Figure 3b). Note how this makes the program malformed, indicated by the disabled "OK" (Figure 3c). If x is nevertheless the variable on which c is to be accessed, this can be confirmed, by selecting x as in Figure 3c. This selection uniquely determines the parent of field c (was: Y), as can be seen from Figure 3d, in which X now appears as unchangeable in column "new". Pressing the (now available) "OK" executes the selected robust edit.

The solution space explorer is also used as a prototype UI for adding and deleting nodes, when attributes need to be (re-)assigned as a result (see Section 5.1). For a discussion of this UI, we refer the reader to Section 8.

6 Edit Intentions

The main problem of the generic solution presented above is that the user may quickly get lost in the solution space (but see Section 8 for an outline of a better UI). Given that in many cases, the editing problems and their solutions will be stereotypical in nature, exposing the user to the full space seems unnecessary. For instance, looking at the edit options of our motivating example from Section 2, we can identify at least the following patterns of robust edit operations:

- Select a valid pair of a record-typed variable and a field accessed on it (option ① in Section 2).
- Select a variable and a new field added to the record that serves as the type of the selected variable, setting the type of the field to fit the context (option ⑥).
- Select a variable and delete the field access (⑤).

Since they express what the user has in mind with an intended edit, we call patterns of this kind *edit intentions*, or *intentions* for short.

Figure 4 shows intention specifications capturing the above informally specified patterns in a DSL designed for this purpose. All three specifications are associated with the struct `VarRef`, of which the use of `x` in `x.a` is an instance; `self` refers to the instance of the struct the intention is applied to. Note that our intention DSL borrows the path expressions from our constraint DSL of Section 3.2, for identifying attributes that may (or must) be re-assigned.

The first specification, named "select ⟨variable.field⟩", is a *modify* intention stating that the node's attribute `var` (cf. Figure 1) and its parent's attribute `field` can be assigned new references. Here, `parent<DotExp>` expresses that the

```
modify "select <variable.field>" {
  self.var
  self.parent<DotExp>.field
}
add "select <variable.newfield>" {
  struct: Fdecl
  assign: self.parent<DotExp>.field
  modify: self.var
}
replace "select <variable> & delete ." {
  become:
    self.parent<DotExp>
    self
  modify: self.var
}
```

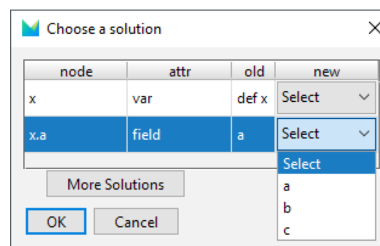
Figure 4. Intention specifications associated with `VarRef`.

parent of `self` must be an instance of `DotExp`; otherwise, the intention cannot be applied.

The second specification "select ⟨variable.newfield⟩" is an *add* intention that creates a new instance of struct `Fdecl` and assigns a reference to this instance to the `field` attribute of the parent of the node on which this intention is invoked. The mandatory attributes of the new field (specifically: `id`, `type`, and `parent`, i.e., the record to which the field is added), are automatically selected as assignable (they must be assigned for the program to be well-structured; cf. Section 3.1). The `modify` clause of this intention expresses that additionally, the `var` attribute may be re-assigned (in our example from a reference to `x` to one to `y`). Note that, in our example, the parent of the new field is uniquely determined by the well-formedness constraint "field access" and follows the type of the variable, meaning that if the user selects a reference to `y` for `self.var`, the new field declaration is added to record `Y`. Also note that, if the user names the new field "a", the name disequality constraint "unique field names" of Figure 2 prevents that it is added to `X`; `x` can therefore not be selected, unless more solutions are requested (see below). The type of the new field is constrained by the type of the dot expression ("dot type"), which is constrained (via "var type") to the type of `z`; however, since these are not declared as re-assignable, the type for the new field is set to `X`, the type of old field `a`. Note that `type` attributes can be declared as assignable, too, but type changes can propagate widely through a program, bordering on type refactorings [30] (see below and Section 9).

The third specification "select ⟨variable⟩ & delete ." is a *delete* intention that re-assigns all attributes referencing the first argument of `become:` (reminiscent of Smalltalk's `become:`) with a reference to the second (in our example, references to the expression `x.a` are replaced with references to `x`). A dynamic check associated with `become:` allows this intention to be applied only if it leaves the ASG well-structured, specifically, if the declared struct of each attribute holding a reference to the dot-expression is a superstruct of `DotExp` and `VarRef`.

Each of these intentions specifies generically which attributes (constraint variables) may or must be re-assigned to perform the associated edit operations; it is understood that all others remain untouched ("frame axiom"). If the corresponding CSP has no solutions, the user can request an expansion of the editable attributes, by branching to the solution space explorer; in fact, for the selection of the assignments necessary to specify the concrete robust edit (the instantiation of the pattern specified by an intention), we currently branch to the solution space explorer anyway, restricting the available selections to those specified by the intention. For instance, applying the edit intention "select ⟨variable.field⟩", the user sees



offering the currently available robust edits instantiating the pattern, and the possibility to request more.

Our edit intentions are somewhat similar to quick-fixes, and also to refactoring tools, in that they encode stereotypical edit operations; however, unlike the latter two, they do not require separate implementations, but are completely driven by existing well-formedness rules (see Section 9 for a comparison with related work). In fact, since our intentions are completely specified using a (fairly simple) DSL, versed users can easily extend the intention catalog shipped with the RPE with their favorite edit intentions.

7 Implementation

So far, we supposed that an ASG and editing intent are mapped to a CSP, which is then submitted to a constraint solver. However, a number of reasons let this procedure appear less than ideal.

- Constraint solving can quickly become very expensive. This is especially so for mostly symbolic domains (which ASGs are) abounding with disjunctions, as introduced by path expressions (see Sections 5 and 9), which force the solver to branch. Especially for large ASGs, with nodes in the millions, the unrolling of path expressions is very hard to confine so that the resulting CSPs can be solved in acceptable time.
- To present the user with *all* available options to complete an editing intent, we need to compute *all* solutions of a CSP. This voids all attempts of solvers to find the first solution quickly — it is the effort required to compute the last that limits the utility of constraint solving for our purpose.

- While all solutions are required, the selection process described in Section 5.2.2 projects these solutions to domains of variables, from which the user sequentially selects. This means that, for any single selection, we do not need the complete set of solutions; a conservative approximation ensuring that all available options for a single attribute are actually supported by a solution will be enough. Once a selection has been made for one attribute, the approximation can be updated for the remaining attributes, until only one attribute remains.

The last finding coincides with the fact that standard constraint solving is divided into two phases: constraint propagation and search [31]. Constraint propagation usually establishes a form of local consistency, essentially by removing values from variable domains that are not supported by any assignment of other variables of the same constraint (see below), while search tries to find a solution in the space (Cartesian product) set up by the so-reduced domains. Note that, while search is generally sufficient to solve a CSP, the reduction of the search space bought by constraint propagation usually expedites search tremendously, and hence contributes much more to constraint solving than it costs. The domains obtained by constraint propagation are close to the conservative approximation that we required above for selecting a value of any single attribute; to be sufficient, however, they would also need to be globally consistent [3].

To compute consistent domains, we build on Mackworth's original n-ary arc consistency algorithm NC [9]. In a nutshell, NC maintains a queue (worklist) W of pairs (C, x), initialized with all constraints C and constraint variables x of a CSP such that x is directly constrained by C. From W, and until W is empty, NC removes a pair (C, x) and from the domain of x, D_x, all values that are not supported by values from the domains $D_{x'}$ of C's other directly constrained variables x'.[9] If this leaves D_x empty, the algorithm terminates with the result that the CSP is inconsistent (no robust edit can be computed for the ASG, intent, and assignable variables). Else, if any values were deleted from D_x, it appends all pairs (C', x') such that $C' \neq C$ and C' directly constrains x, and $x' \neq x$ and x' is directly constrained by C'.

When W is finally empty and more than one variable domain has more than one value ("non-singleton domain"), NC continues by bisecting one of these domains, and starting over for each of the resulting CSPs. Upon termination of this recursive procedure, each terminal process provides n sets of solutions, where each set is comprised of one of the n values of the remaining non-singleton domain (if present), and the values from the singleton domains.

[9] A value of a variable of a constraint is supported if the constraint can be satisfied by assigning its other directly constrained variables values from their domains.

Straightforward though it may seem, application of NC in our domain is threatened by the high arity of constraints involving path expressions. Indeed, as noted in Section 5, in a constraint C mapped from a well-formedness rule involving a path expression like `self.field.type`, if the variable representing `self.field` can take on any of n values, each value leads to a different `type` constraint variable, increasing the arity of C by n. This would not only make checking the support for values of variables from C more expensive, but would also increase the length of the worklist considerably. However, due to the disjunctive nature of path expressions (only one of the n variables is selected, and hence constrained, by any concrete path), all values for variables reached in a constraint exclusively via the dereferencing of others are always supported, as long as it may be the case that the variables are not selected (in which case they are free to adopt any value). For instance, if the domain of (the constraint variable representing) `self.field` is {`a`, `b`}, any value in the domain of both `a.type` and `b.type` is supported, for `a.type` by assuming the value `b` for `self.x`, and for `b.type` by assuming `a`. The reader versed in constraint solving will recognize this as a form of constructive disjunction [6], for which efficient filtering algorithms exist (ours builds on insights detailed in [8]). Note that, should the constraint propagation performed by NC reduce the domain of a dereferenced variable to a singleton, the referenced variable becomes uniquely selected, and hence will become subject to domain reduction.

The same reasoning applies to our `*`"-operator (see Section 3.2): While it introduces an arbitrary length path expression, and with it may select a large number of additional variables, the domains of these variables cannot be reduced (for the same reasons as above), so that they need not be added to the worklist. However, the variables may still need to be visited for finding a support of other variables' values; in these cases, the disjunctive nature of path expressions allows the search to be aborted once one disjunct has led to a supporting value, so that the transitive closure implied by the "`*`"-operator does not have to be explored in full.

After achieving local consistency, NC would continue by bisecting a non-singleton domain and recursing. However, given that we need the user to select a solution (Section 5.2.2), this appears premature — instead, we let the user select a value from a non-singleton domain, and recurse with the domain set to this value. To be sure that any selection from a domain computed by NC will actually lead to a solution, this procedure needs global consistency [3].

However, when achieving global consistency, we pay the price for dropping from local consistency the variables of path expressions reached via dereferencing other variables. In untoward constellations (when their domains have not been restricted by other constraints), this may bring out the exponential nature of global consistency, and stretch the user's patience accordingly. In these cases, we can still push global consistency to a background task, running for

completion while the user inspects the solution space as suggested in Section 5.2.2, only now with domains underapproximated by the achieved local consistency. The user's selections continuously reduce the size of the problem, making global consistency cheaper to achieve; in the worst case, the user has chosen a value that does not lead to a solution of the CSP, in which case the process backtracks.

The constraint solving required for the application of robust editing intentions (as described in Section 6) is implemented in the exact same way; due to its inherent restriction of the variables that are subject to constraint solving (i.e., that may be assigned values), achieving global consistency is not a problem here.

We have implemented our RPE on top of the meta-programming system MPS. Our implementation replaces MPS's use of scope rules for determining possible selections of references with the more general constraint satisfaction approach described in this paper, which allows other well-formedness rules (including type rules) to be exploited for the same purpose. In fact, this resolves one of the most criticized (among members of our group) oddities of MPS, its reliance on three different mechanisms for enforcing scope rules, type rules, and other rules of well-formedness (each coming with its own DSL), while at the same time allowing us to evaluate scope rules "reversely", for computing legal locations for newly added program elements. For reasons outlined above, we use a custom implementation for achieving local and global consistency; apart from liberating us from the limited expressiveness of standard constraint solvers (e.g., none known to us supports transitivity as required by our "*" operator), it also allows us to propagate constraints "on the fly", that is, without first setting up a complete constraint graph, and then transforming parts of it to a CSP submitted to a constraint solver. This greatly reduced the overhead of our constraint approach.

Standard MPS already comes with a notion of intentions as atomic actions initiated by the user to modify an existing program in a predefined way. However, these intentions are programmed imperatively, and make no guarantees with respect to well-formedness. Our intentions add to those offered by MPS, and are invoked in exactly the same way; in fact, entering the solution space explorer at any given editing location is also implemented as an (MPS) intention.

8 Discussion

Although still not widely used by programmers, projectional editors are helpful for non-programmers writing programs in DSLs [37]. In fact, their support for flexible and mixed syntax facilitates replicating a domain's inherent notations, and their support for language modularity helps with the efficient development of DSLs [35]. For programmers using GPLs, these advantages may be outweighed by a clumsier editing process (cf. Footnote 2); yet, projectional editors are increasingly being recognized as ideal launch pads for advanced language services (see [14] and also Section 9), and indeed, building tool support for robust editing without the aid of projectional (or structure) editors seems hard [14].

We make no claims regarding the practicality of the prototype UI of our implementation of an RPE, which we presented in Sections 5 and 6. Specifically, we understand that it will be hard for the user to relate nodes and properties shown in the solution space explorer to the different places in a program where they occur, but it is easy to conceive of an alternative UI that comes with a robust editing mode, in which the program remains browsable/navigable as usual, but in-place editing is limited to the alternative editing options currently collated in the solution space explorer, until a robust edit has uniquely been identified and the robust editing mode is left again. The primitive edits yet to be decided on can be maintained in a list for quick reference and navigation (not unlike the problems view we see in traditional IDEs), and exploring the solution space can be enhanced with undo functionality; yet, given that undo does not work reliably even for standard MPS, this still seems some way to go.

Robust edits can be very powerful. For instance, by moving `def z = x.a` (from the motivating example) to `module A` (expressed as a primitive edit by assigning the corresponding node's `parent` attribute a reference to the node of `module A`), the computed complementing edits will suggest to move the nodes on which the moved nodes depends via scope constraints as well. While many would equate this change with a refactoring (especially if offered as an intention "move ⟨node⟩"), we uphold here the original definition of refactoring, which would require guarantees of the change being behavior-preserving (and not only well-formedness preserving; see Section 9). In fact, programming may be seen as an alternating sequence of behavior-altering and behavior-preserving changes, and hence (since behavior preservation requires and produces well-formedness), as an alternating sequence of robust edits and refactorings: With our introduction of RPEs, we have complemented refactoring tools as automated editing engines with an engine for performing the well-formedness preserving changes between refactorings; together, they cover the full editing process.

While we presented robust projectional editing as a transformation from well-formed programs to well-formed programs, the exploration of the solution space that we described and implemented suggests that only a small part of the program is mapped to a CSP, meaning that only for this part, preservation of well-formedness is guaranteed. While this does not invalidate our claims (robust editing will still not introduce malformedness), the requirement that the program must be globally well-formed before a robust edit is unnecessarily strong if one accepts that the well-formedness preservation guarantee of robust projectional editing extends only to the parts covered by the editing. Note in particular that this allows the program to be merely pre-structured before a robust edit is performed, meaning that robust edits

may be alternated with ordinary ones — the same constraints that would be used for computing robust edits are then used to check well-formedness after the edit. However, as soon as the solution space associated with a robust edit extends to attributes contributing to mere pre-structuredness or malformedness, the robust edit will ensure that these conditions are resolved. Robust editing may thus also be used to repair malformed programs.

9 Related Work

Our design and implementation of an RPE seems related to the semantic foundations of program editors recently postulated by Omar et al. [14]. Specifically, our pre-structured ASGs capture programs with syntactic holes, and representing these holes as variables of a CSP allows us to reason about, and complete, incomplete programs. Furthermore, as has been argued elsewhere [25], the same constraint-based captures of well-formedness can also be used for repairing malformed programs [26] and, by additionally casting behavior-critical relations between program elements (nodes) to constraints, for refactoring programs [22]. By contrast, editor services offered by contemporary IDEs each rely on their own implementation, and hence on a duplication of logic, causing the obvious consistency and maintenance problems.

The propagation of change in presence of (OCL) constraints has been a continuing theme in the works of Egyed and Reder (see. e.g., [4, 18, 19]). However, their implementations appear to be based on explicitly maintained dependency graphs (along which concrete changes are propagated), and not on constraint solving techniques, as our work. It is unclear how this addresses circular dependencies; also, without constraint propagation (arc consistency), it will be very expensive to compute a complete set of legal (robust) edit alternatives for selection, as we do.

Robust editing is somewhat related to ad-hoc refactoring, or "refactorings without names" [22, 29], and intentions are loosely related to the refactoring specification of Refacola [27]. The main technical differences are that here, we do not distinguish between a constraint generation and a constraint solving phase and do not employ a standard constraint solver for the latter as we did in our previous works, but rather apply local and global consistency techniques directly on the internal representation of a program, and involve the programmer in the search that is normally performed by the constraint solver. This allows us to use the same set of constraints and technical infrastructure for checking well-formedness and for computing robust edits, avoiding the redundancy of (constraint-based) refactoring tool and (standard) compiler from which all our constraint-based refactoring tools suffered. Robust editing is also related to code completion, as described, for instance, in [17].

MPS, on which we have implemented RPE, has native mechanisms for enforcing scope rules, type rules, and other conditions of well-formedness [34]. Scope rules, which are called scope constraints in MPS, compute the set of available targets for a given reference (that is, those program elements that are "in scope"), allowing the editor to enforce "well-scopedness" in addition to pre-structuredness as described in Section 3.1. When the user attempts to establish a reference, only the targets in this set can be chosen from. The same computation is used to check well-scopedness of a program after other changes: for this, the sets of available references are re-computed, and existing references that are not members of the "available" sets are flagged as a scope errors. By contrast, type rules do *not* constrain the construction of the program[10]; instead, they are evaluated as part of static checking, flagging errors "after the fact". The same holds for other conditions of well-formedness. By contrast, our implementation of RPE uses the same mechanisms uniformly for reference selection and checking; in fact, if the only edits allowed are robust edits, well-formedness checking will only be needed for loading programs that have not been written using our RPE. Because of this uniform treatment, we also have a single DSL for expressing scope rules, type rules, and other conditions, unlike MPS, which offers different DSLs for each purpose. Importantly, our approach also supports the on-demand creation of not yet existing reference targets based on the existing constraint-based description of scoping rules. MPS's native approach does not support this; instead, on-demand target creators have to be implemented manually, duplicating the logic in the scoping rules.

The integration of scope rules and type rules underlying our RPE has also been suggested in [11, 33], in the context of name binding. As far as type members are concerned, name binding is also touched on by the type constraints of [15, 30]; indeed, as we have shown elsewhere [23, 25], the constraint-based coverage of type rules can be generalized to cover name binding. However, name binding is not an issue in the context of projectional editing (see Section 3), so that we do not use this feature here.

The translation of path expressions (e.g., ↑1.`field.type`; see Section 5) is a common problem of mapping "object" constraint languages (i.e., constraint languages that assume objects and reference semantics of attributes;,e.g., OCL) to plain value-based constraint solvers, namely when attributes on the path can be changed by constraint solving (so that subsequent attributes are accessed indirectly). One solution is to use `element` constraints [5, 27], which model the indirection with an array access. Alternatively, if the used constraint solver supports this, reference attributes can be mapped to uninterpreted functions (so that `x.field.type` becomes `type(field(x))`), letting the solver find values for function applications [2]. A solution that will work for any constraint solver, yet may require clumsy constraint rewriting, is to wrap indirect attribute access in conditional constraints, as suggested by [16, 23]. In any case, indirection has a heavy

[10] but note that access of type members is subsumed by the scope rules

performance penalty, particularly if the domain of the indirection (i.e., dereferenced) variable cannot be sufficiently restricted. With our custom implementation of constraint solving, we avoid this problem for ensuring local consistency (see Section 7); more generally, and since the problem coincides with an exploded solution space, we counter it through the introduction of intentions (Section 6), which limit generic variability to the intended ("good") parts.

10 Conclusion

While the program analyses underlying editor services such as code completion or refactoring tools require programs to be well-formed (or otherwise produce erroneous results), editing may leave a program malformed, even when projectional, or structure, editors are used. With our introduction of *robust edits*, we raise editing itself to a language service that complements intended edits with others that are required to preserve well-formedness. We show how, from a user perspective, complexity is reduced by the introduction of *editing intentions*, named patterns of stereotypical editing operations declaratively specified using a small DSL; technically, we tame it by involving the user in the computation of a robust edit, by interleaving local and global consistency techniques with the user's selection of primitive edit operations. Unlike that of most other language services in use today, our approach is fully declarative, using the same specification for checking well-formedness, type inference, and computing robust edits, avoiding consistency and maintenance problems by construction.

Acknowledgments

This work was supported by DFG grant STE 906/5-1.

References

[1] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of projectional editing: A controlled experiment. In *Proc. of the 2016 24th ACM International Symposium on Foundations of Software Engineering*. ACM, 763–774.

[2] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings*. 337–340.

[3] Rina Dechter. 1992. From Local to Global Consistency. *Artif. Intell.* 55, 1 (May 1992), 87–107.

[4] Alexander Egyed. 2006. Instant consistency checking for the UML. In *28th International Conference on Software Engineering (ICSE 2006)*. 381–390.

[5] Pascal Van Hentenryck. 1989. *Constraint satisfaction in logic programming.* MIT Press.

[6] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. 1998. Design, Implementation, and Evaluation of the Constraint Language cc(FD). *J. Log. Program.* 37, 1-3 (1998), 139–164.

[7] R. Lämmel. 2017. *Software languages: Syntax, semantics, and metaprogramming.* Springer. To appear. Book's website: http://www.softlang.org/book.

[8] Olivier Lhomme. 2003. *An Efficient Filtering Algorithm for Disjunction of Constraints.* Springer, 904–908. http://dx._doi.org/10.1007/978-3-540-45193-8_76

[9] Alan K. Mackworth. 1977. On Reading Sketch Maps. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'77)*. Morgan Kaufmann Publishers Inc., 598–606.

[10] Raul Medina-Mora and Peter H. Feiler. 1981. An Incremental Programming Environment. *IEEE Trans. Software Eng.* 7, 5 (1981).

[11] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015. Proceedings*. 205–231.

[12] David Notkin. 1985. The GANDALF project. *Journal of Systems and Software* 5, 2 (1985). DOI: http://dx.doi.org/10.1016/0164-1212(85)90011-1

[13] Object Management Group. 2014. *Object Constraint Language Version 2.4.* Object Management Group.

[14] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages, SNAPL 2017*. 11:1–11:12.

[15] Jens Palsberg and Michael I. Schwartzbach. 1994. *Object-oriented type systems.* Wiley.

[16] Nils Przigoda, Robert Wille, and Rolf Drechsler. 2016. Ground setting properties for an efficient translation of OCL in SMT-based model finding. In *Proc. MODELS 2016*. ACM, 261–271.

[17] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM Conference on Programming Language Design and Implementation, PLDI '14*. 419–428.

[18] Alexander Reder and Alexander Egyed. 2012. Computing repair trees for resolving inconsistencies in design models. In *IEEE/ACM Intl. Conf. on Automated Software Engineering, ASE'12, 2012*. 220–229.

[19] Alexander Reder and Alexander Egyed. 2012. Incremental Consistency Checking for Complex Design Rules and Larger Model Changes. In *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012. Proceedings*. 202–218.

[20] Thomas W. Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. In *First ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. ACM. DOI: http://dx.doi.org/10.1145/800020.808247

[21] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. In *OOPSLA 2006*. ACM.

[22] Friedrich Steimann. Constraint-Based Refactoring. *ACM TOPLAS* (????). to appear.

[23] Friedrich Steimann. 2015. From well-formedness to meaning preservation: model refactoring for almost free. *Software and System Modeling* 14, 1 (2015), 307–320.

[24] Friedrich Steimann. 2015. None, One, Many - What's the Difference, Anyhow?. In *1st Summit on Advances in Programming Languages, SNAPL 2015*. 294–308.

[25] Friedrich Steimann. 2015. Refactoring Tools and Their Kin. In *Grand Timely Topics in Software Engineering - International Summer School GTTSE 2015, Tutorial Lectures (Lecture Notes in Computer Science)*, Jácome Cunha, João Paulo Fernandes, Ralf Lämmel, João Saraiva, and Vadim Zaytsev (Eds.), Vol. 10223. Springer, 179–214. DOI: http://dx.doi.org/10.1007/978-3-319-60074-1

[26] Friedrich Steimann, Jörg Hagemann, and Bastian Ulke. 2016. Computing repair alternatives for malformed programs using constraint attribute grammars. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*. 711–730.

[27] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. 2011. A Refactoring Constraint Language and Its Application to Eiffel. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference. Proceedings*. 255–280.

[28] Friedrich Steimann and Bastian Ulke. 2013. Generic Model Assist. In *Model-Driven Engineering Languages and Systems - 16th International*

Conference, MODELS 2013. Proceedings. 18–34.

[29] Friedrich Steimann and Jens von Pilgrim. 2012. Refactorings without names. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12*. 290–293.

[30] Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.* 33, 3 (2011), 9.

[31] Edward P. K. Tsang. 1993. *Foundations of Constraint Satisfaction*. Academic Press.

[32] Bastian Ulke, Friedrich Steimann, and Ralf Lämmel. Partial Evaluation of OCL Expressions. In *Model-Driven Engineering Languages and Systems - 20th Intl. Conference, MODELS 2017. Proceedings. To appear.*

[33] Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016*. 49–60.

[34] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org.

[35] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. 2017. Lessons learned from developing mbeddr: a case study in language engineering with MPS. *Software & Systems Modeling* (2017), 1–46.

[36] Markus Voelter and Sascha Lisson. 2014. Supporting Diverse Notations in MPS'Projectional Editor.. In *GEMOC@ MoDELS*. 7–16.

[37] Markus Voelter, Jos Warmer, and Bernd Kolb. 2015. Projecting a modular future. *IEEE Software* 32, 5 (2015), 46–52.

[38] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*. Springer, 41–61.