Robust Real-Time Synchronization between Textual and Graphical Editors

Oskar van Rest[1,2], Guido Wachsmuth[1,3], Jim R.H. Steel[2],
Jörn Guy Süß[2], and Eelco Visser[1]

[1] Delft University of Technology, The Netherlands,
o.f.vanrest@student.tudelft.nl, g.h.wachsmuth@tudelft.nl,
visser@acm.org
[2] The University of Queensland, Australia,
jsteel@uq.edu.au, jgsuess@itee.uq.edu.au
[3] Oracle Labs, Redwood Shores, CA, USA

Abstract. In modern Integrated Development Environments (IDEs), textual editors are interactive and can handle intermediate, incomplete, or otherwise erroneous texts while still providing editor services such as syntax highlighting, error marking, outline views, and hover help. In this paper, we present an approach for the robust synchronization of interactive textual and graphical editors. The approach recovers from errors during parsing and text-to-model synchronization, preserves textual and graphical layout in the presence of erroneous texts and models, and provides synchronized editor services such as selection sharing and navigation between editors. It was implemented for synchronizing textual editors generated by the Spoofax language workbench and graphical editors generated by the Graphical Modeling Framework.

1 Introduction

Modeling languages such as Behavior Trees [3,17] or QVT Relational [18] provide both textual and graphical concrete syntax. Textual and graphical editors for such languages need to synchronize textual representations, graphical representations, and underlying models. During this synchronization, layout in textual and graphical representations needs to be preserved.

Textual editors generated by textual modeling frameworks such as TEF [19] and Xtext [8] synchronize only on user request. Embedded textual editors based on TEF synchronize on open and close [20]. Xtext-based editors synchronize on save [16]. This breaks the interactive nature of integrated development environments (IDEs), where editors provide a wide variety of language-specific services such as syntax highlighting, error marking, code navigation, content completion and outline views in real-time, while their content is edited. Furthermore, those editors can only synchronize valid models and tend to break either textual or graphical layout. TEF-based editors ignore textual layout by design. Xtext-based editors typically preserve textual layout, but tend to break layout in graphical editors once identifiers change.

Robust real-time synchronization of textual and graphical editors is mainly prevented by current text-to-model transformation practice, where model elements are temporarily deleted and recreated during parsing, existing persisted
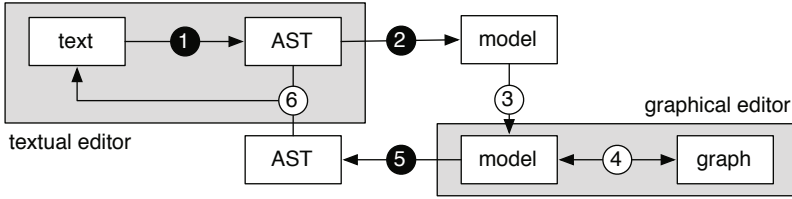
Fig. 1. Steps involved in synchronizing textual and graphical editors: ❶ Parsing, ❷ tree-to-model transformation, ③ model merge, ④ edit policy, ❺ model-to-text transformation, ⑥ pretty-printing. Steps marked black support error recovery. Steps marked white support layout preservation.

models are ignored and overwritten by new models, and error recovery is limited. In this paper, we propose a new approach which is outlined in Fig. 1. To synchronize textual changes with a model, the text is ❶ parsed into an abstract syntax tree, which is ❷ transformed into a model. The resulting model is ③ merged with the model in a graphical editor, which invokes an edit policy to ④ update its graphical representation of the model. To synchronize graphical changes with a text, the edit policy ④ changes the underlying model, which is ❺ transformed into a tree. The resulting tree is ⑥ merged with the tree in the textual editor and turned back into text. The approach was implemented for synchronizing textual editors generated by the Spoofax language workbench [13] and graphical editors generated by the Graphical Modeling Framework for the Eclipse IDE. We applied this approach to Behavior Trees. Fig. 2 shows the textual and graphical editor, which both share the same Behavior Tree model.

We proceed as follows. We first describe a mapping from grammars to metamodels and the corresponding transformations ❷❺ between trees and models. In Sect. 3, we discuss error recovery in steps ❶❷❺. In Sect. 4, we elaborate on the preservation of textual and graphical layout in steps ③④⑥. In Sect. 5, we present our case study on the development of synchronizing editors for Behavior Trees. Finally, we discuss related work in Sect. 6.

2   Tree-to-Model and Model-to-Tree Transformations

The textual syntax definition is the starting point of our approach. In this section, we present a mapping from textual syntax definitions to metamodels and a corresponding bidirectional mapping between abstract syntax trees conforming to the textual syntax definition and models conforming to the generated metamodel. We start with abstract mappings which need to be adapted for concrete formalisms. We then discuss such an adaptation using the examples of Spoofax' syntax definition formalism SDF [9,26], its name binding language NaBL [14], and EMF's metamodeling formalism Ecore [24].

2.1   Mapping Textual Syntax Definition to Metamodel

We start with minimalistic grammar and metamodeling formalisms. In these formalisms, grammars, metamodels and models are represented as terms. Fig. 3 shows
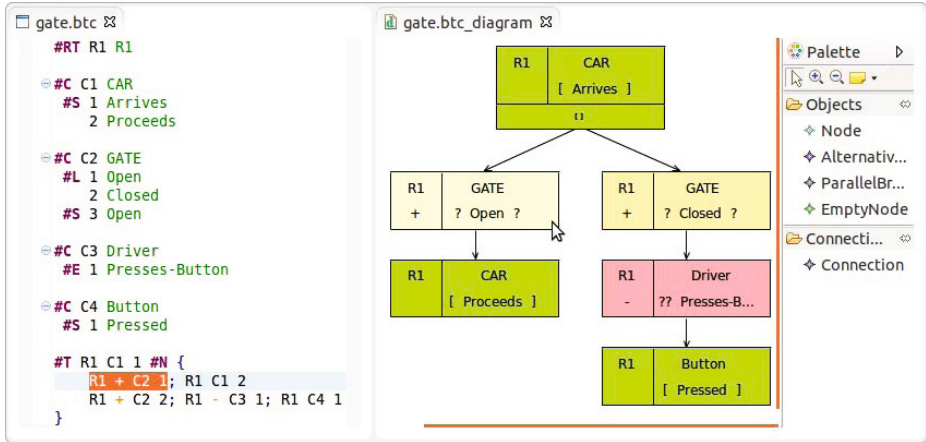
Fig. 2. Behavior Tree model in a textual editor (left) and in a graphical editor (right). Both editors edit the same model and synchronize changes with each other.

the corresponding signatures. These signatures are of the form c : T → s with c a constructor for sort s and T a declaration of the number and types of arguments of c. The mapping is specified in Fig. 4 by rewrite rules expressed in Spoofax' transformation language Stratego [1,11]. These rules are of the form r : t1 → t2 where s with r the rule name, t1 and t2 first-order terms, and s a *strategy expression*. A rule applies to a term if the term matches t1 and s succeeds, resulting in the instantiation of t2.

Grammars, metamodels, and models. A grammar consists of a lexical syntax definition, a context-free syntax definition, and a list of namespace specifications (Fig. 3, line 1). Both lexical and context-free syntax are defined by productions, which are grouped by the sorts they define (l. 2). Productions and sorts are named, and each production provides a list of symbols (l. 3). A symbol is either a character class (typically used to define lexical sorts), a string, a reference to a lexical sort, or a reference to a context-free sort (ll. 4-7). References are named (first ID), refer to a sort by name (second ID), and might come with a postfix operator for options, lists, or optional lists. References to lexical sorts can be involved in name bindings, either as definition or use sites of a name in a namespace (ll. 8-11). This integration of name binding into productions is similar to Xtext's approach. But in contrast to Xtext, we decouple namespaces from sorts and allow them to be hierarchically structured.

A metamodel consists of a list of types, which are either primitive data types, enumerated data types, abstract classes, or concrete classes (ll. 16-21). Type names are qualified, providing a simple packaging mechanism. Both kinds of classes consist of a list of qualified parent class names, defining the inheritance hierarchy, and a list of features. We distinguish attributes, references, and containments (ll. 22-24). Each feature is named, refers its type by qualified name, and defines a lower and upper bound (ll. 25-26).

```
1    Grammar: List(Sort)*List(Sort)*List(NSpace)  → Grammar
2    Sort   : ID*List(Prod)                        → Sort
3    Prod   : ID*List(Symbol)                      → Prod
4    Chars  : List(Char)                           → Symbol
5    Literal: String                               → Symbol
6    LSort  : ID*ID*Binding*Operator               → Symbol
7    CfSort : ID*ID*Operator                       → Symbol
8    None   :                                        Binding
9    DefSite: ID                                   → Binding
10   UseSite: ID                                   → Binding
11   NSpace : ID*List(ID)                          → NSpace
12   None   : Operator
13   Option : Operator
14   List   : Operator
15   OptList: Operator
```

```
16   MM     : List(Type)                  → Metamodel
17   DType  : QID                         → Type
18   Enum   : QID*List(Literal)           → Type
19   AClass : List(QID)*QID*List(Feature) → Type
20   CClass : List(QID)*QID*List(Feature) → Type
21   Literal: ID                          → Literal
22   Attr   : ID*QID*Bounds               → Feature
23   Ref    : ID*QID*Bounds               → Feature
24   Contain: ID*QID*Bounds               → Feature
25   QID    : ID*ID                       → QID
26   Bounds : INT*UnlimitedINT            → Bounds
```

```
27   M      : Object              → Model
28   Obj    : Opt(URI)*QID*List(Slot) → Object
29          : Value               → Slot
30          : Opt(Value)          → Slot
31          : List(Value)         → Slot
32   Data   : String              → Value
33   Link   : URI                 → Value
34   Contain: Object              → Value
```

Fig. 3. Signatures for grammars (top), metamodels (center), and models (bottom)

A model is represented as a single root object (l. 27). An object consists of an optional URI, the qualified name of the class it instantiates, and a list of slots (l. 28). A slot may hold a single value or a list of values, where a value is either an instance of a data type represented as a string, a link to an object represented as the URI of this object, or a contained object (ll. 29-34). Slots do not refer to features. Instead, we assume an immutable order of the features of a class, which links slots of an object to the features of its class.

Lexical Syntax. We are not interested in the inner structure of lexical tokens and represent them as basic data at the leaves of abstract syntax trees. We can keep the same basic data in models. Thus, we map lexical sorts from a grammar to data types in a metamodel (Fig. 4, ll. 7-14). Predefined data types (enumerations and primitives) are provided by the metamodel formalism and the condition lex2qid ensures that user-defined data types are only generated when no corresponding predefined data type exists. When a lexical sort defines only a finite number of literals, an enumeration is generated (sort2enum). Only when sort2enum fails, we try to generate a primitive with sort2dtype

```
1   grammar2mm:
2     Grammar(lex*, cf*, ns*) → MM([ty1*, ty2*, ty3*])
3     where
4       <filter(sort2enum <+ sort2dtype)> lex* ⇒ ty1* ;
5       <mapconcat(sort2classes)> cf*        ⇒ ty2* ;
6       <map(ns2class)> ns*                  ⇒ ty3*
```

```
7   sort2enum:
8     Sort(name, prod*) → Enum(<lex2qid> name, <map(prod2lit)> prod*)
9
10  prod2lit: Prod(_, [Literal(name)]) → Literal(name)
11
12  sort2dtype: Sort(name, _) → DType(<lex2qid> name)
13
14  lex2qid: name → QID("lex", name) where <not(predefined)> name
```

```
15  sort2classes:
16    Sort(name, prod*) → [AClass([], QID("cf", name), [])|class*]
17    where
18      <map(prod2class(|name)))> prod* ⇒ class*
19
20  prod2class(|parent):
21   Prod(name, sym*) → CClass([parent|parent*], Q("ast", name), feat*)
22   where
23     <filter(symbol2parent)> sym*  ⇒ parent* ;
24     <filter(symbol2feature)> sym* ⇒ feat*
25
26  symbol2feature:
27    LSort(label, sort, None(), op) → Attr(label, ty, <op2bounds> op)
28    where
29      <predefined <+ user−defined> sort ⇒ ty
30
31  symbol2feature:
32    CfSort(lbl, sort, op) → Contain(lbl, QID("cf", sort), <op2bounds> op)
33
34  op2bounds: None()    → Bound(1, 1)
35  op2bounds: Option()  → Bound(0, 1)
36  op2bounds: OptList() → Bound(0, Unbound())
37  op2bounds: List()    → Bound(1, Unbound())
```

```
38  ns2class:
39    NSpace(name, ns*) → AClass(<map(ns2qid)> ns*, QID("ns", name), [])
40
41  ns2qid: name → QID("ns", name)
42
43  symbol2parent: LSort(_, _, DefSite(nspace), _) → QID("ns", nspace)
44
45  symbol2feature:
46    LSort(label, sort, DefSite(_), op) → Attr(label, ty, <op2bounds> op)
47    where
48      <predefined <+ user−defined> sort ⇒ ty
49
50  symbol2feature:
51    LSort(label, _, UseSite(ns), op) → Ref(label, QID("ns", ns), bounds)
52    where
53      <op2bounds> op ⇒ bounds
```

Fig. 4. Rewrite rules defining a grammar-to-metamodel transformation in Stratego

(in the first condition for grammar2mm, <+ encodes a deterministic choice). To
avoid name conflicts, we organize generated data types in a package lex.

Context-free Syntax. Abstract syntax trees represent the structure of sentences. We can express such trees also as models. Therefore, the metamodel
needs to capture the structural rules of the context-free syntax. We achieve

```
 1  tree2model: t          → M(<term2obj>)
 2  term2obj  : c#(t*)      → Obj(<def−uri>, QID("ast", c), <map(term2slot)> t*)
 3  term2slot : None()      → None()
 4  term2slot : Some(t)     → Some(<term2slot> t)
 5  term2slot : t*          → <map(term2slot)> t*
 6  term2val  : t           → Data(t) where is−string; not(ref−uri)
 7  term2val  : t           → Link(<ref−uri>)
 8  term2val  : t           → Contain(<term2obj> t) where is−compound
 9
10  model2tree: M(obj)                       → <obj2term> obj
11  obj2term  : Obj(_, QID("ast", c), s*)    → c#(<map(slot2term)> s*)
12  slot2term : None()                       → None()
13  slot2term : Some(val)                    → Some(<slot2term> val)
14  slot2term : val*                         → <map(slot2term)> val*
15  val2term  : Data(val)                    → val
16  val2term  : Link(uri)                    → <name−of> uri
17  val2term  : Contain(obj)                 → <obj2term> obj
```

Fig. 5. Rewrite rules defining corresponding tree-to-model and model-to-tree transformations in Stratego

this by generating classes from context-free sorts and productions (ll. 15-24). To avoid name conflicts, we organize them in separate packages cf and ast. For each context-free sort, we generate an abstract class (sort2classes). For each production of this sort, we generate a concrete class subclassing the abstract class (prod2class). Features are generated from the symbols of the production (ll. 26-32). We generate an attribute for each lexical sort (first rule). The type of this attribute is derived from the lexical sort. For each context-free sort, we generate a containment reference (second rule). Bounds of generated features depend on operators (ll. 34-37). Options get a lower bound of 0, while all other symbols get a lower bound of 1. Lists get an unlimited upper bound, while all other sorts get an upper bound of 1.

Name Binding. In our minimalistic grammar formalism, namespaces and sorts are separate concepts. Thus, namespaces impose their own class hierarchy on the generated metamodel. For each namespace, we generate an abstract class which subclasses its parent namespaces (ll. 38-43). When a production defines a definition site of a name, the concrete class generated from this production needs to subtype the namespace of the definition site. Therefore, symbol2parent collects the namespaces of definition sites. At definition sites, the generated feature is the same as for ordinary lexical sorts (ll. 45-48). At use sites, a reference to the namespace is generated instead (ll. 50-53).

2.2   Bidirectional Mapping between Trees and Models

We specify a bidirectional mapping between trees and models as a pair of unidirectional mappings tree2model and model2tree in Fig. 5.

To transform a tree into a model, we transform its term representation into an object (tree2model). This is done by decomposing the term into its constructor c and subterms t*. The constructor is used to identify the corresponding class in the operator and the subterms are transformed into slots. When a term is the
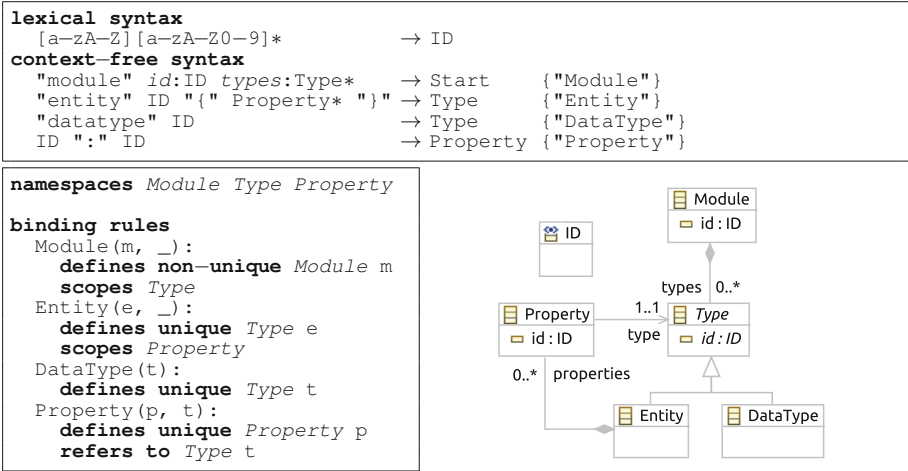
```
lexical syntax
  [a—zA—Z][a—zA—Z0—9]*        → ID
context—free syntax
  "module" id:ID types:Type*   → Start     {"Module"}
  "entity" ID "{" Property* "}" → Type      {"Entity"}
  "datatype" ID                → Type      {"DataType"}
  ID ":" ID                    → Property {"Property"}
```

```
namespaces Module Type Property

binding rules
  Module(m, _):
    defines non—unique Module m
    scopes Type
  Entity(e, _):
    defines unique Type e
    scopes Property
  DataType(t):
    defines unique Type t
  Property(p, t):
    defines unique Property p
    refers to Type t
```

Fig. 6. Syntax definition in SDF (top), name binding rules in NaBL (left) and generated Ecore metamodel (right) for an entity language.

definition site of a name, we expect `def-uri` to provide a URI for it. Otherwise, it should yield `None()`. The first rule of `term2val` transforms strings (the leaves of a tree) into (one of) the slot's value(s). The rule only works if the string is not the use site of a name. The second rule covers such use sites, by generating a link with a URI. We expect `ref-uri` to provide the URI of a bound name. Otherwise, it should fail. The third rule of `term2val` transforms compound terms into contained objects.

The rules for `model2tree` mirror the rules for `tree2model`. We expect `name-of` to yield the name which establishes the binding to the linked object.

2.3   Connecting Spoofax and EMF

In Spoofax, lexical and context-free syntax are defined in SDF [9,26]. Name binding and scope rules are defined separately in NaBL [14]. From these definitions we generate metamodels in Ecore, EMF's metamodeling formalism [24]. Fig. 6 shows syntax definition, name binding rules and generated metamodel for a small data modeling language.

SDF and NaBL differ from the minimalistic grammar formalism in several ways. First, naming conventions are different. Since symbols are only optionally labeled in SDF, we generate missing labels either from sorts or from referred namespaces. We use annotated constructor names as production names. Since these are not required to be unique in SDF, we generate unique names where needed. Second, SDF supports special injection and bracket productions,which we model by inheritance. Third, SDF provides additional kinds of EBNF-like operators and allows to apply them not only to sorts, but on any symbol. We introduce intermediate sorts to break down such applications. Finally, NaBL

separates name binding rules from productions. We weave productions and name binding rules based on their constructors.

Ecore differs from the minimalistic metamodel formalism as well. The only relevant differences are order and uniqueness of many-valued features. Since text is sequential, we generate ordered features. While references and containments are inherently unique in Ecore, we generate non-unique attributes. In a post-processing step, we simplify the generated metamodel. We fold linear inheritance chains, merge classes which share all their subclasses, and pull common features from subclasses into their parent class.

For the mapping between trees and models, we apply the previously shown transformations.Additionally, we provide a thin, generic Java layer which can convert between models as Spoofax terms and models as EMF objects.

3   Error Recovery

Error recovery is crucial for real-time synchronization between editors. Furthermore, it allows for persisting erroneous models using the textual syntax. We distinguish three kinds of errors which affect editor synchronization. *Parse errors* and *unresolved names* are discovered in the textual editor when the text is parsed to an AST which is afterwards statically analyzed. *Graphical syntax errors* occur in the graphical editor when a model does not satisfy lower bound constraints of its metamodel. Graphical editors relax this constraint to allow for incremental modeling. More specific, semantic errors do not affect synchronization and error marking for such errors is allowed in either the textual or graphical editor, or both.

Parse Errors. Modern IDEs parse text with every change that is made to it, ensuring rapid syntactic and semantic feedback as a program is edited. As text is often in a syntactically invalid state as it is edited, parse error recovery is needed to diagnose and report parse errors, and to construct a valid AST for syntactically invalid text. Therefore, Spoofax has strong support for parse error recovery [4]. It introduces additional recovery productions to grammars that make it possible to parse syntactically incorrect text with added or missing characters. These rules are automatically derived from the original grammar. Spoofax' parsing algorithm activates these rules only when syntax errors are encountered and uses layout information to improve the quality of recoveries for scoping structures, while still ensuring efficient parsing of erroneous text. This approach avoids the loss of AST parts when a correct text is changed into an incorrect one, which is crucial for real-time synchronization.

Unresolved names. Spoofax resolves names after parsing with an algorithm which is based on declarative name binding and scoping rules [14]. The algorithm is language-independent, handles multiple files, and works incrementally, which allows for efficient re-analysis after changes. During intermediate editing stages, not all references may be resolved. Fig. 7 illustrates this with a simple data model. It contains a property `title` of type `Strin`, which cannot be resolved.
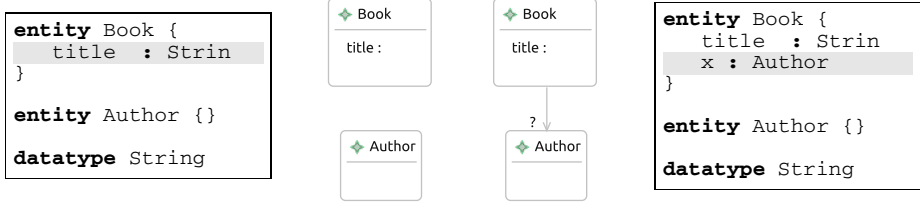
Fig. 7. Recovery from a name resolution error and from a graphical syntax error

We recover from such errors during tree-to-model transformation (step ❷). Spoofax provides special URIs for unresolved references. When we discover such a URI, we do not fill the corresponding slot in the model. GMF handles such underspecified models and visualizes model elements with unfilled slots. In the example from Fig. 7, the property appears in the graphical editor without any type. The user can specify the missing type either by continue typing or by choosing the type in the properties view of the graphical editor.

Graphical syntax errors. During graphical editing, newly added model elements are typically underspecified. Since graphical editors do not enforce completion, a user might first create a number of such underspecified elements before she starts to complete them. To recover from such errors, the model-to-tree transformation needs to handle incomplete models (step ❺). A simple fix would be to map unfilled slots to empty strings in the AST. Step ⑥ would add these empty strings at positions where the parser expects text for the missing element. The parser recovers from such errors, but might report the error at a different position, confusing the user. To overcome this problem, the model-to-tree transformation creates textual default values for unspecified attributes and references and ignores elements with unspecified containments.

Both attributes and references are represented by strings in text. If they are unspecified upon model-to-text transformation, we generate a default value that conforms to the lexical syntax. For example, if an integer is expected, we take default value 0, while if a string is expected, we take default value x (cf. Fig. 7). Note that in case of a reference, it is important not to choose an existing name, since this will connect every new model element to an existing one. The generation of default values introduces unresolved names and possibly semantic errors as well. These errors are marked until they are resolved by completing underspecified elements. Users may also switch to textual editing in the meantime, and resolve the errors by typing. The solution can be further improved by allowing users to specify default values in the syntax definition, as one may not prefer the 'default' defaults. Unspecified containments should ideally not be permitted by graphical editors. In the graphical Behavior Trees editor (Sect. 5), for example, we automatically create both an atomic sequence and a contained node upon using the node tool. However, this is not possible if multiple subtypes are allowed, in which case the user needs to manually indicate the type of the contained element. Therefore, we ignore elements with unspecified containments during model-to-tree transformation. This means that users are required to complete such an element
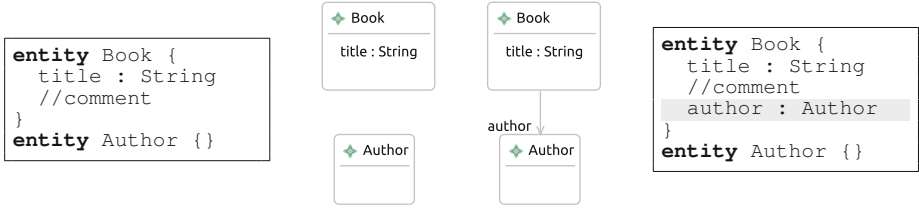
```
entity Book {
  title : String
  //comment
}
entity Author {}
```

```
entity Book {
  title : String
  //comment
  author : Author
}
entity Author {}
```

Fig. 8. Textual layout preservation and pretty-printing in reaction to a new property

before switching to the textual editor, or the element will be destroyed upon the
next text-to-model transformation.

4   Layout Preservation

Textual layout consists of comments and whitespace, while graphical layout con-
sists of positions and sizes of graphical elements. This information needs to be
preserved during editor synchronization. Our approach to layout preservation is
based on merging in both directions (steps ③⑥). New ASTs or models are com-
pared against their old version to calculate differences between them. Differences
are then merged into the relevant representation, which causes the representation
to be incrementally updated with changes from the other editor.

Textual Layout Preservation. Spoofax supports textual layout preservation
for refactorings [5]. To achieve this, it combines origin tracking with pretty-
printing. We reuse this feature to preserve textual layout when propagating
changes from the graphical editor to text. Origin tracking relates nodes in an
AST with text fragments. This information is propagated by transformations.
It is lost when we transform a tree into a model, but it is still available in
the AST of the textual editor. Pretty-printing considers this old AST and a
new one generated by model-to-tree transformation. It compares both ASTs and
preserves text corresponding to unchanged parts. Fragments corresponding to
removed parts are removed from the text. New AST nodes are pretty-printed and
inserted into the text. For this purpose, Spoofax generates pretty-printing rules
from the syntax definition, which can be enhanced with user-defined rules [4].

Fig. 8 shows an example for the data modeling language that involves pretty-
printing. First, a reference of type Author is added to the entity Book in the
graphical editor. A new object is added to the underlying model and positional
information for the connection anchors is added to the notation model. Model-
to-tree transformation yields a new AST. Each of its subterms will match with
a term in the old AST, except for the term corresponding to the new reference.
This term is pretty-printed and inserted into the text. Comments and whitespace
in the surrounding text are preserved.

The approach works in the presence of any type of syntactic or semantic error.
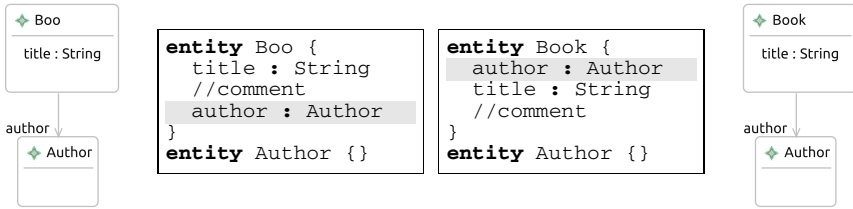However, it fails during a graphical cut-and-paste operation. Cutting destroys

Fig. 9. Graphical layout preservation in reaction to a changed order of properties

the corresponding textual element and its associated layout. The element is recreated upon pasting, but its original layout is lost.

Graphical Layout Preservation. Spoofax re-parses text once it changes. Tree-to-model transformation turns the new AST into a model which is merged with the model from the graphical editor. We rely on EMF Compare [28,2] for comparing and merging models. Since old and new model will typically show much resemblance, difference calculation is very precise and changes that require merging are very small.

An example is given in Fig. 9 where we change the order of entities in the text. The text is parsed and a new AST is created that shows the reordering of two subterms. Tree-to-model transformation yields a new model which is compared against the old one. The only difference is a change in the order of the owned references of the `Module` object. We merge this into the old model, which result in a reordering of a list. Since the order of the entities is not graphically represented, GMF keeps the notation model and the diagram unchanged.

Fig. 9 shows another example in which we change the identifier of an entity. Many model merging approaches use identifiers of objects for matching. When an identifier changes, objects are no longer matched resulting in a deletion and re-creation. Layout of the deleted object is lost. EMF Compare takes not only identifiers but all slots of an object into account. It typically matches renamed elements and layout information can be preserved.

This is also shown in Fig. 9. Here, we change the name of entity `Boo` into `Book`. This change is reflected in the new AST and the new model. Since the `name` slots of the old and new `Entity` object show much resemblance and since both objects contain the same property, the objects match. During merging, only the value of the `name` attribute changes. During synchronization with the graphical editor (step ④), no information is added to the notation model, but the label corresponding to the attribute is re-rendered to show the new name.

The approach works in the presence of any type of semantic or graphical syntactic error. However, sometimes we cannot recover from a parse error, in which case only a partial AST is created. Synchronization will then destroy the graphical elements corresponding to the erroneous text region. Upon resolving the error, the graphical elements are recreated, but their original layout is lost. Furthermore, similarly to textual layout preservation, graphical layout preservation fails during textual cut-and-paste operations.

5   Case Study: Behavior Trees

A behavior tree (see Fig. 2 and Fig. 10 for examples) is a formal, tree-like graphical form that represents behavior of individual or networks of entities [7]. The Behavior Trees (BT) language has formed the base of Behavior Engineering (BE), an approach to systems development that supports the engineering of large-scale dependable software intensive systems [17].

Tooling support for BT initially focused on graphical editors only [27,22]. Recently, the language was extended with a formal textual syntax [3] and TextBE, a textual editor combined with a visualizer based on EMFText and SVG Eclipse, was introduced [17]. Although textual editing greatly reduced the time to create behavior tree models, TextBE is still limited in that it visualizes only on-save, which inhibits the expected interaction in an IDE, prevents manual layout of the graphical representation, which affects use cases like printing and sharing seriously, and does not provide navigation means between textual and graphical representation, which inhibits fast visual search. We applied our approach to BT in order to create an integrated textual and graphical editor. The editors synchronize in real-time, allow for manual and automated textual and graphical layout, preserve layout during synchronization, and support selection sharing to navigate between both representations.

Fig. 10 illustrates the robustness of the text-to-model transformation. First, the identifier of state Closed changes from 2 into 21. As a consequence, references to this state become unresolved and the label in the graphical model vanishes. Next, a semicolon is added after the node which results in a parse error. Though, a new model element appears since the semicolon indicates a following node. When we continue typing R1 C3 1, the graphical node is incrementally
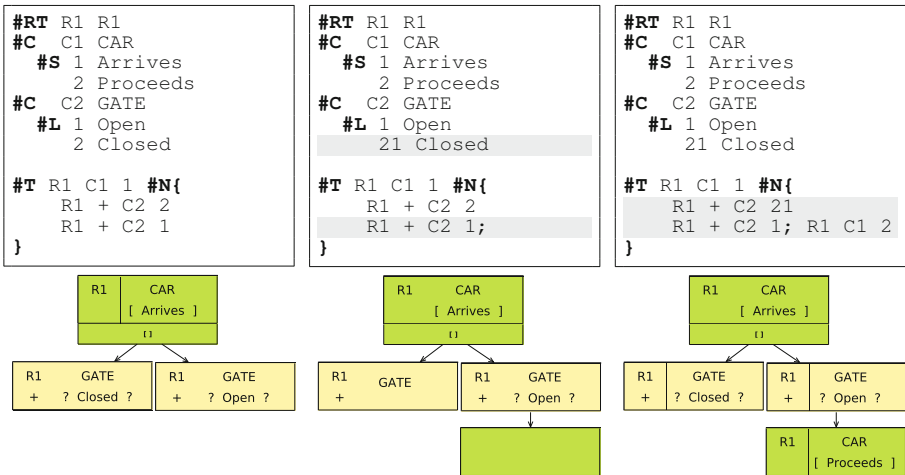

Fig. 10. Graphical layout preservation and auto-layout during textual editing and in the presence of parse errors and unresolved references.
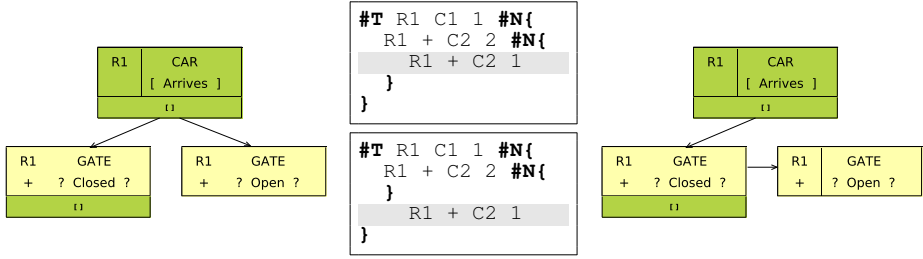
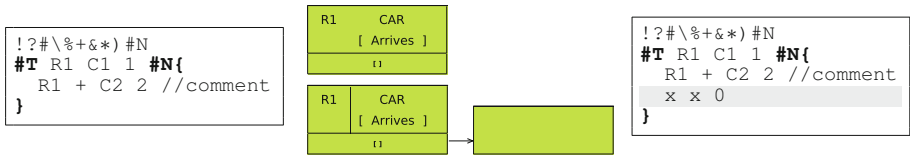**Fig. 11.** Graphical layout preservation during textual drag-and-drop

Fig. 12. Textual layout preservation and pretty-printing during graphical editing in the presence of both graphical and textual syntax errors

built up while its original assigned position is maintained. Finally, we update the broken reference by typing an additional 1. This resolves the reference such that the graphical representation shows the same label as before. Fig. 11 illustrates a more drastic change, where text is dragged from an inner scope and dropped in an outer scope. Graphically, connections between nodes change accordingly, while all positions of nodes are preserved. Fig. 12 shows the text before and after a new node is created in the graphical editor. Although mandatory features are not yet specified, we already obtain a textual representation of the node. The pretty printer automatically indents the node, while layout information consisting of whitespace and a comment is preserved. This all works in the presence of an erroneous text region (cf. !?#%+&*)#N) which is also preserved.

6   Discussion

There are two classes of tools that support textual and graphical model editing. *Textual modeling frameworks* such as Xtext [8], MontiCore [15], EMFText [10], TCS [12], and TEF [19] support textual editing by parser generation and a generic tree-to-model mapping. First, we discuss their specification approach and then we discuss their support for editor synchronization, error-recovery and layout preservation. Similar to our approach, Xtext provides a grammar-based formalism and generates metamodels. However, name binding constructs provided by the formalism are limited. Scope and import rules, which can be declaratively defined in NaBL, need to be implemented in Java instead. This is

a limitation of the other frameworks as well. MontiCore provides a formalism for describing both a grammar and a metamodel. While we automatically derive a set of abstract classes and an inheritance hierarchy from the textual syntax definition, MontiCore allows users to manually specify those. This provides more flexibility to influence the resulting metamodel. TEF requires the user to specify both grammar and metamodel, which is very redundant. EMFText and TCS take the opposite approach and start from a metamodel. TCS requires the user to specify templates, which are then used for parsing and pretty-printing. With EMFText, user-defined templates are optional, since it generates default ones based on the UML Human-Usable Textual Notation [23].

Of all the frameworks, only TEF merges textual and graphical models, while the others only inherit limited synchronization capabilities from EMF and GMF, where models are synchronized on save. Saving overwrites the previous model and breaks references from the notation model. A GMF edit policy then tries to repair references, which often leads to deletions followed by re-creations of notations and a loss of layout. All frameworks except TEF support textual layout preservation. However, in on-save synchronization multiple model changes are merged into the text at once, making the merging process imprecise such that layout is not always preserved correctly. All frameworks have only limited error recovery capabilities such that switching between textual and graphical editing is only possible if models are not broken. However, on-save synchronization does not introduce the problem of layout being destroyed when elements are temporarily lost due to parse errors or cut-and-paste operations. Our approach could possibly be improved by providing additional means to maintain such layout, or by delaying synchronization until parse errors don't result in partial ASTs anymore and cut-and-paste operations are completed.

Projectional editors as provided by MPS [6] or Intentional [21] support different views and different concrete syntax projections which are automatically synchronized since they all share the same abstract syntax model. Editing directly affects the abstract syntax, similar to graphical editing. However, typical textual operations such as indentation or rearranging are not possible, but can be simulated to a certain degree. Another notable tool that provides mappings between text and models is Enso [25], which requires the user to provide both a grammar and a metamodel that need to be kept consistent.

7   Conclusion

This paper presented an approach for robust real-time synchronization between textual and graphical editors. It recovers from errors during synchronization and preserves textual and graphical layout during editing, even in the presence of errors. It allows for a new type of highly interactive editors and has successfully been applied to the Behavior Trees modeling language.

References

1. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. SCP 72(1-2), 52–70 (2008)
2. Brun, C., Pierantonio, A.: Model differences in the Eclipse Modeling Framework. In: UPGRADE, IX (April 2008)
3. Colvin, R., Hayes, I.J.: A semantics for Behavior Trees using CSP with specification commands. SCP 76(10), 891–914 (2011)
4. de Jonge, M., Nilsson-Nyman, E., Kats, L.C.L., Visser, E.: Natural and flexible error recovery for generated parsers. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 204–223. Springer, Heidelberg (2010)
5. de Jonge, M., Visser, E.: An algorithm for layout preservation in refactoring transformations. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 40–59. Springer, Heidelberg (2012)
6. Dmitriev, S.: Language oriented programming: The next programming paradigm (2004)
7. Dromey, R.G.: From requirements to design: Formalizing the key steps. In: SEFM, pp. 2–11 (2003)
8. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: OOPSLA, pp. 307–309 (2010)
9. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN 24(11), 43–75 (1989)
10. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
11. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. SoSyM 9(3), 375–402 (2010)
12. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE, pp. 249–254 (2006)
13. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA, pp. 444–463 (2010)
14. Konat, G., Kats, L., Wachsmuth, G., Visser, E.: Declarative name binding and scope rules. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 311–331. Springer, Heidelberg (2013)
15. Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 286–300. Springer, Heidelberg (2007)
16. Mülder, A., Nyßen, A.: TMF meets GMF. Kombination textueller und grafischer Editoren. Eclipse Magazin 3, 74–78 (2011) (in German)
17. Myers, T.: TextBE: A textual editor for behavior engineering. In: ISSEC (2011)
18. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1 (January 2011)
19. Scheidgen, M.: Integrating content assist into textual modelling editors. In: Modellierung, pp. 121–131 (2008)
20. Scheidgen, M.: Textual modelling embedded into graphical modelling. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 153–168. Springer, Heidelberg (2008)

21. Simonyi, C.: The death of computer languages, the birth of intentional programming. In: NATO Science Committee Conference (1995)
22. Smith, C., Winter, K., Hayes, I.J., Dromey, R.G., Lindsay, P.A., Carrington, D.A.: An environment for building a system out of its requirements. In: ASE, pp. 398–399 (2004)
23. Steel, J., Raymond, K.: Generating human-usable textual notations for information models. In: EDOC, pp. 250–261 (2001)
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2009)
25. van der Storm, T., Cook, W.R., Loh, A.: Object grammars: Compositional & bidirectional mapping between text and graphs. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 4–23. Springer, Heidelberg (2013)
26. Visser, E.: A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam (August 1997)
27. Wen, L., Colvin, R., Lin, K., Seagrott, J., Yatapanage, N., Dromey, R.G.: "Integrare", a collaborative environment for behavior-oriented design. In: Luo, Y. (ed.) CDVE 2007. LNCS, vol. 4674, pp. 122–131. Springer, Heidelberg (2007)
28. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE, pp. 54–65 (2005)