Sheffield Hallam University

A declarative approach to heterogeneous multi-mode modelling languages.

CLARK, Tony <http://orcid.org/0000-0003-3167-0739>

Available from Sheffield Hallam University Research Archive (SHURA) at:

https://shura.shu.ac.uk/12069/

This document is the Published Version [VoR]

Citation:

CLARK, Tony (2014). A declarative approach to heterogeneous multi-mode modelling languages. In: COMBEMALE, Benoit, DEANTONI, Julien and FRANCE, Robert, (eds.) GEMOC 2014: globalization of modeling languages: proceedings of the 2nd International Workshop on The Globalization of Modeling Languages co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (Models '14). CEUR Workshop Proceedings (1236). Tilburg University, 33-42. [Book Section]

Copyright and re-use policy

See http://shura.shu.ac.uk/information.html

Sheffield Hallam University Research Archive
http://shura.shu.ac.uk

A Declarative Approach to Heterogeneous Multi-Mode Modelling Languages

Tony Clark

Department of Computer Science, Middlesex University, London, UK
t.n.clark@mdx.ac.uk

Abstract. This paper proposes a declarative approach to multi-mode heterogeneous DSLs based on term rewriting. The paper presents a data model and algorithm for processing syntax structures. It has been validated by an implementation that supports a range of languages. The paper includes an example language that supports both game construction and execution.

1 Introduction

Domain Specific Languages (DSLs) [25, 17] are motivated by the need to define languages that match specific use-cases, as opposed to General Purpose Languages (GPLs). Whilst GPLs are usually supported by standard text editors, DSLs, by their nature, often contain a range of more exotic syntax elements that are arguably better supported by syntax-aware editors. This has led to the development of a range of technologies to support DSL development and that generate tools for each DSL. Where the DSL is limited to text, languages such as EMFText [8], MontiCore [15, 16], TCS [10], and XText [6], MPS and Spoofax [11] allow a DSL to be quickly and conveniently defined and the associated tooling generated. These technologies are mainly based on grammarware [12] that integrate language parsers with editors in order to achieve a workbench. Many of the technologies integrate static and dynamic analysis of the resulting DSL. These technologies have become quite mature and the term Language Workbench [7] has been coined to describe this type of engineering tool.

Whilst languages used for programming or scripting tend to be exclusively text-based, modelling languages have included a much wider palette of elements. UML for example, has a number of sub-languages that are based on graphs, but also includes text in the form of OCL and action languages. Relatively few technologies support the definition and tooling of DSLs containing graphical syntax elements. Exceptions include Eugenia [13, 14], GMF [9], MetaEdit+ [23].

There has been increasing attention to *heterogeneous* (mixing graphical and textual notations) [1, 21, 5, 20]. Intentional Software and MPS are both developing tools that support *projectional editors* [22]. A recent model-based approach to mixing text and graphical languages is described in [2] that uses projectional editing techniques over a model. Whilst most of the reported work agrees on the general principles and proposed approaches, there has been little work on providing a concrete heterogeneous approach. In addition, most language use-cases

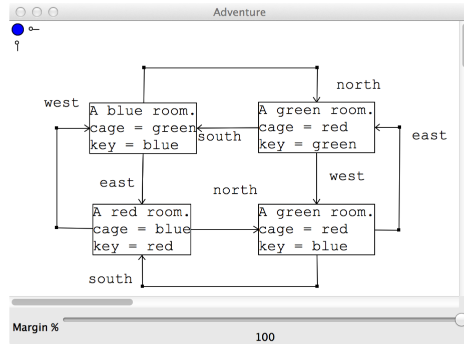


Fig. 1: Defining the Game

involve multiple modes, minimally *definition* and subsequent *use*. Other modes include *debugging* and using a language from the perspective of different stakeholders. Most DSL technologies do not support multi-mode interaction within the same tool-set.

This paper presents a novel declarative approach to the definition and associated tooling for heterogeneous multi-mode DSLs. The contribution is to propose that simple term rewriting can be used as the basis of this approach. This paper describes an algorithm that is suitable for this purpose and demonstrates how the declarative approach can be used to define a multi-mode heterogeneous DSL for building and playing a game. The approach has been validated by implementing the algorithm and the associated tool can be downloaded with examples.

2 Example

Consider a game that involves a collection of rooms that are connected by corridors. A room is either empty or contains a locked cage. The cage is painted red, green or blue. Inside the cage is a painted key. A key can be used to unlock a cage of the same colour and get the key inside. The player starts off in a room with a red key. The aim of the game is to visit all the rooms and unlock all the cages. Figure 1 shows the definition of a dungeon using the language editor for game construction. Rooms are created as nodes and corridors as labelled edges. The text in a room-node shows the colour of the room, the colour of a cage and the colour of the key in the cage. The blue dot at the top-left corner of the tool is used to access a room-creation menu. Edges between room-nodes are created by dragging the mouse from a source node to the target (a menu is used to select a direction). When a room-node is created, its colour and contents are uninitialised: the mouse is used to select from pre-defined colours for the room, cage and key.

The language operates in two modes: creation and play, it is possible to switch between the modes by pressing `p` and `c` on the keyboard. Figure 2 shows play mode. The player starts in the blue room with a red key. The player makes a move by pressing the first letter of the direction on the keyboard. Since the

player does not have a green key they must move from the starting room; they press `n` to go north and arrive at a green room with a red cage. The player can open the cage since their key matches the cage colour. This is done by pressing `u` on the keyboard. Finally, the player goes back south.

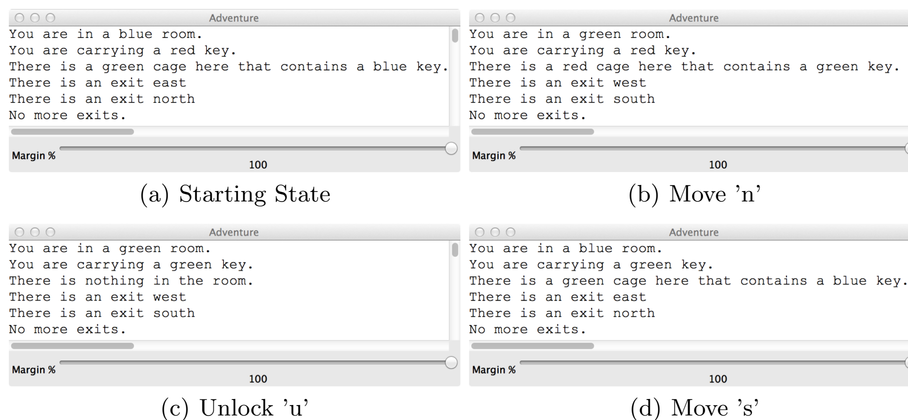
The game shows a number of features of the projectional editor. Interaction with the language can be moded; in this example there are two modes, but in general there can be any number. The abstract syntax can be projected on to graphs and text. In addition, language features can be created by menus made available as blue-dots. Figure 1 shows a blue dot that is used to create room-nodes, but in general a language may offer many different types of item. Figure 2 shows that the state of the game is projected to become formatted text. Figure 3 shows how the editor that is generated from the language definition supports creation of language elements: (a) creation of a new room element; (b) selection of a room colour; (c) selection of a type of edge between rooms.

3 Declarative Language Definition

The approach uses a simple term representation for both concrete and abstract syntax, and uses term rewriting as the technology to support all language modes. Section 3.1 describes the data representation and an rewriting algorithm, section 3.2 describes how concrete syntax is represented as trees in normal forms, and section 3.3 describes how rules are used to define the terms used to represent abstract syntax.

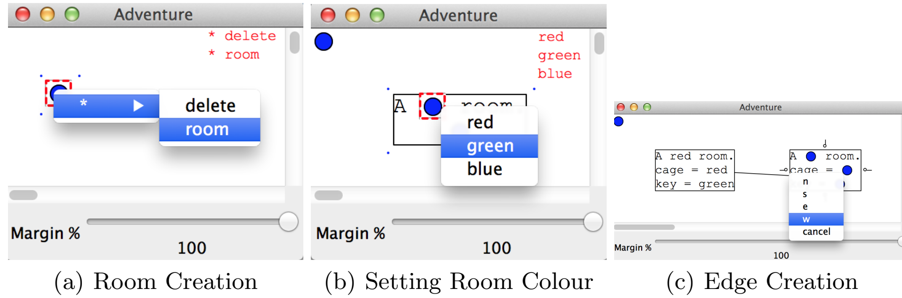
3.1 Syntax Trees and Transformations

The DSL editor manages a syntax tree. A tree is in one of a number of forms: *atomic* in which case it is a string, number, char or boolean; *term* in which case it has the form `f[i1,...im](t1,...,tn)` where `f` is the term-*functor* which is a



(a) Starting State (b) Move 'n' (c) Unlock 'u' (d) Move 's'

Fig. 2: Playing the Game



(a) Room Creation (b) Setting Room Colour (c) Edge Creation

Fig. 3: Editor Interactions

name, `i1,...,im` is the identity of the term, and `ti` are sub-trees. The identity of a term is used to associate a term with layout information. Where the identity of a term is not important it is omitted and assumed to be unique.

The editor works by creating an initial tree referred to as *abstract syntax*. The abstract syntax tree is *transformed* into a concrete syntax tree by applying pattern directed transformation rules. Concrete syntax is a tree whose term-functors are pre-defined.

The editor displays a concrete syntax tree, and then waits for a user-event. The set of user-events is pre-defined; each event is mapped on to a term whose functor designates the type of event: `key`, `selected`, `drag`, *etc*. The abstract syntax tree `t` and the new event-term `e` are combined into a new term whose functor is $\leftarrow$, written `t` $\leftarrow$ `e`. The transformation process is repeated, allowing rules that process `e` to transform `t` appropriately.

Transformation rules have the form `p` $\rightarrow$ `e` where `p` is a pattern and `e` is an expression. A pattern is a tree that contains variables. A variable is just a name that starts with an upper-case letter. An environment `E` associates variables with either trees or lists of trees, and associates functors with functions. An environment that contains no lists is applied to a pattern to create a tree `E(p)=t`. For example, `{X` $\mapsto$ `10}(f[0](X))=f[0](10)`.

Patterns may be repeated. Such a pattern must be nested within a parent pattern and is followed by ... The informal meaning of such a pattern is '*match as many elements as possible then continue*'. For example, the pattern `f[0](p...,10)` will match a term whose functor is `f`, whose identity is `0` and whose children end with `10`, providing that all other children each match pattern `p`.

An environment that applies to a pattern containing repeated sub-patterns will contain mappings between variables and lists of trees. For example, `{X` $\mapsto$`[1,2,3]}(f(X...,10))=f(1,2,3,10)`. Repeated patterns need not be atomic as in the following example: `{X` $\mapsto$`[1,2,3]}(f(g(X)...,10))=f(g(1),g(2),g(3),10)`.

Expressions are patterns that can refer to local variables and local function definitions. A local function definition is a collection of rules that are selectively applied to part of a tree. For example, if `L=10` and `f(A)=g(A)` are local definitions then the term `x(f(L))` in the context of these definitions will be transformed to `x(g(10))`.

```
1 proc transition(tree,locals,transformations,reductions,table) {
2   while(true) {
3     tree := transform(transformations,locals,tree);
4     let tree' := transform(reductions,tree) such that tree' ∈ 𝒩
5     table := display(tree',table);
6     tree := get_event(tree',tree,table);
7   }
8 }
9 fun transform(rules,locals,tree) {
10   while ∃ p → e ∈ rules and ∃ E ∈ ℰ such that I ⊆ E and E(p) = tree do {
11     tree := E(e)
12   }
13   let f[i1,...,in](t1,...,tn) = tree
14   tree' = f[i1,...,in](t'1,...,t'n) where t'i = transform(rules,locals,t'i)
15   if tree' = tree
16   then return tree
17   else return transform(rules,locals,tree')
18 }
```

Fig. 4: Projectional Editor Algorithm

A rule `r` is a pattern and an expression `p` $\rightarrow$ `e`. Given a tree, `t`, and some local definitions `L`, `r` is applicable when there is an environment `E` that contains `L` for which `E(p)=t`. The result of applying the rule to `t` is then `E(e)`.

The behaviour of the editor is defined in figure 4. The procedure `transition` performs a loop that transforms the abstract to the concrete syntax tree, displays the concrete syntax tree and then waits for a user event. The arguments of `transition` are: the abstract syntax tree, the local definitions, two sets of rules `transformations` and `reductions`, and a table that maps term identities to layout information.

Line 3 uses the transformation rules to change the current abstract syntax tree. This allows events to be change the state of the tree. Line 4 uses the reduction rules to transform the tree into a normal form, *i.e.*, a member of the set of trees $\mathcal{N}$ that can be drawn by the editor. The resulting concrete syntax `tree'` is displayed by the editor in line 5 using the table to remember the layout information on each loop within the procedure `transition`.

Line 6 waits for a user event. Such an event will occur with respect to the concrete syntax, so the table is used to make the correspondence between elements in `tree'` and in `tree`, resulting in a new abstract syntax tree of the form `t` $\leftarrow$ `e`.

The procedure `transform` is used to apply rules to a tree in the context of some local definitions. Lines 11-13 continually select a rule that is applicable and updates the tree. Once there are no more applicable rules, line 15 transforms the children of the tree. If any children have changed then the process is repeated (line 18) otherwise no more rules are applicable to any part of the tree and it is returned (line 17).

3.2 Normal Forms

The set $\mathcal{N}$ of normal forms contains trees whose functors and structure correspond to concrete syntax elements that can be drawn on a screen and that

can respond to user events. Different editors may define different sets of normal forms, for example a text-only editor may only support trees that correspond to string layout, whereas a graph editor may only support trees representing collections of nodes and edges.

This paper uses a game to explain the key features of the projectional editor approach to heterogeneous DSLs. The normal forms used by the game are:

`atom` Any atomic value is a normal form.

seq`(t1,...,tn)` The sub-tree normal forms are displayed in sequence.

nl Produce a new-line.

graph`(etypes,nodes-and-edges)` The graph is displayed on the screen and supports selection, new edge, movement, resize, and mouse click events. The identities of the nodes and edges are used to ensure that the layout is consistent; therefore, a node with a fresh identity will cause a new node to appear on the screen. The `etypes` define the permissible edge types, and `nodes-and-edges` is a mixed sequence of nodes and edges.

edge-types`(t1,...,tn)` each `ti` is of the form `type(source,target)` where `type` is the type designator for edges that can be drawn from nodes of type `source` to nodes of type `target`.

node`(ntype,i,display)` where `ntype` designates the type of node, `i` is the identity of the abstract syntax element represented by the node (for passing back events on this node), and `display` is a normal form that is displayed when this node is drawn.

edge`(source,sdec,target,tdec,label)` where `source` and `target` are the source and target node identities, `sdec` and `tdec` are the edge-end decorations for the source and target, and `label` is a label on the edge.

vbox`(pelements)` A vertical box of elements that are all of the form `l(e)` where `l` is one of the layout designators: **centre**; **left**; **right**.

3.3 Abstract Syntax

The editor must start with an initial abstract syntax structure. This is defined by an **abstract** clause in the language definition that corresponds roughly to a type definition for abstract syntax trees. The clause consists of a number of rules of the form `name` → `element` where `element` is one of: a term of the form `f(e1,...,en)` where each `ei` is an element; **str** in which case the element denotes an editable string in the concrete syntax; `e*` where `e` is an element in which case the abstract syntax denotes a sequence of `e`'s of arbitrary length, and is manifest in the concrete syntax in the form of a *hole* that can be selected and incrementally extended via a menu; a disjunction `e1 | e2` which will manifest itself in the concrete syntax as a hole associated with a menu that allows the user to choose between filling the hole with `e1` or `e2`. For example:

```
1 abstract {
2   numbers → number*
3   number  → zero | add(number)
4 }
```

represents an abstract syntax tree that is generated from the first rule (`numbers`). Since the first rule is `number*` we do not know how many instances of `number` to generate, so a hole is displayed allowing the user to generate a `number` followed by another hole, or to delete the hole (completing the sequence). A `number` also generates a hole allowing the user to choose between replacing the hole with a `zero` or a tree containing another number. Although the display of the holes is fixed, the actual representation of terms of the form `add(add(zero))` will depend on the reduction rules that map it into a normal form.

4 Game Implementation

This section describes the game implementation in terms of the abstract syntax definition, the locals, the transformation rules and the reduction rules. The abstract syntax for the game is:

```
1 abstract {
2   game   → game(construct,map(rooms(room*),exits),player)
3   room   → room(colour,empty | cage)
4   colour → red | green | blue
5   cage   → cage(colour,colour) }
```

The game is initially in `construct` mode which means that it will be displayed as a graph and allow new rooms and exits to be added. The map contains an extensible sequence of rooms and empty `exits` and `player` terms. A room has a colour and is either empty or contains a cage. A cage term contains two colours, one for the cage-lock and the other for the key in the cage.

The locals defines the edge types used between room-nodes and a function called `exits-from` that is used to map a room id and a list of all exits from all rooms, to just the exits from the designated room:

```
1 locals {
2   E = edge-types(n(room,room),s(room,room),e(room,room),w(room,room))
3   exits-from(I,Exits) =
4     case Exits {
5       exits → nil
6       exits(exit(D,I,_),Exit...) → cons(D,exits-from(I,exits(Exit...)))
7       exits(_,Exit...) → exits-from(I,exits(Exit...)) } }
```

The transformation rules are used to handle user events and use pattern matching to dispatch on the state of the game:

```
1  transform {
2    game(_,map(rooms(Room[I](Colour,Contents),R...),Exits),_) ← ↓p →
3      game(play,map(rooms(Room[I](Colour,Contents),R...),Exits),player(I,red))
4    game(_,map(Rooms,Exits),Player) ← ↓c → game(construct,map(Rooms,Exits),Player)
5    game(play,map(Rooms,exits(X1...,exit(n,S,T),X2...)),player(S,Carrying)) ← ↓n →
6      game(play,map(Rooms,exits(X1...,exit(n,S,T),X2...)),player(T,Carrying))
7    game(play,map(Rooms,exits(X1...,exit(s,S,T),X2...)),player(S,Carrying)) ← ↓s →
8      game(play,map(Rooms,exits(X1...,exit(s,S,T),X2...)),player(T,Carrying))
9    game(play,map(Rooms,exits(X1...,exit(e,S,T),X2...)),player(S,Carrying)) ← ↓e →
10     game(play,map(Rooms,exits(X1...,exit(e,S,T),X2...)),player(T,Carrying))
11   game(play,map(Rooms,exits(X1...,exit(w,S,T),X2...)),player(S,Carrying)) ← ↓w →
12     game(play,map(Rooms,exits(X1...,exit(w,S,T),X2...)),player(T,Carrying))
13   game(play,map(rooms(R1...,Room[I](C,cage(C-Col,K-Col)),R2...),X),player(I,C-Col)) ← ↓u →
14     game(play,map(rooms(R1...,Room[I](C,empty),R2...),X),player(I,K-Col))
15   game(construct,map(R,exits(E...)),Player) ← new-edge(Type,S,T) →
16     game(construct,map(R,exits(exit(Type,S,T),E...)),Player)
17 }
```

If the user presses the `p` key at any time (line 2) then the game changes state to `play`. If the user presses `n` when the game is in play (line 5), and if there is an exit north from the player's current location `S` then the abstract syntax tree transforms into a new state where the player's current location is `T`. The player can unlock a cage using key `u` (line 13). Finally, if the game is in `construct` mode and the user drags an edge between two graph nodes, then the message `new-`**edge**`(t,s,t)` is send to the tree causing a new exit to be added to the map (lines 15,16). The reduction rules transform the current state of the game into a normal-form ready for display by the editor:

```
1  reduce {
2    game(play,map(rooms(R1...,Room[I](Col,Contents),R2...),exits(Exit...)),player(I,Carry)) →
3      player2str(Carry,Col,Contents,exits-from(I,exits(Exit...)))
4    game(construct,map(rooms(R...),exits(X...)),_) → graph(E,room2n(R)...,exit2e(X)...)
5    player2str(Carry,Col,Contents,X) →
6      seq('You are in a ',Col,' room.',nl,carry(Carry),contents(Contents),exits(X))
7    carry(Key-Colour) → seq('You are carrying a ',Key-Colour,' key.',nl)
8    contents(empty) → seq('There is nothing in the room.',nl)
9    contents(cage(C-Col,K-Col)) →
10     seq('There is a ',C-Col,' cage here that contains a ',K-Col,' key.',nl)
11   exits(nil) → 'No more exits.'
12   exits(cons(Exit,Exits)) → seq('There is an exit ',Exit,nl,exits(Exits))
13   exit2e(Exit[I](Type,S,T)) → edge[I](S,none,T,arrow,label['direction',I](target,Type))
14   n → 'north'
15   s → 'south'
16   e → 'east'
17   w → 'west'
18   room2n(Hole[I](H)) → node['new-room',I](new-room,I,H)
19   room2n(Room[I](Col,Stuff)) →
20     node['room',I](room,I,vbox['b',I](centre(seq('A ',Col,' room.')),centre(Stuff)))
21   red   → 'red'
22   green → 'green'
23   blue  → 'blue'
24   empty → 'empty'
25   cage(Cage-Colour,Key-Colour) → seq('cage = ',Cage-Colour,nl,'key = ',Key-Colour)
26 }
```

Lines 1 - 4 show the two rules that detect whether the game is being constructed or played. If played, then the game state is translated into text. If constructed then the game state is translated to a graph.

5 Implementation

The editor described in this paper has been implemented in the programming language Racket and used to define a range of heterogeneous languages. An implementation pack accompanies this paper[1]. The pack includes a Mac disk image `stand-alone-editor.dmg` of the editor implementation, several saved languages (*.xml) and the source code of the language definitions `language-definitions.rkt`. Once you have installed the editor, navigate to the `bin` directory and start the tool before dragging any of the xml files onto the editor pane to load up the language definition. The pack includes the game language definition, an example adventure, a use-cases implementation of hotel booking and a library class diagram. See the language definitions for more details.

[1] `http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/projectional_editor_demo.zip`

6 Conclusion and Future Directions

This paper has proposed a declarative approach to multi-mode heterogeneous DSLs. The approach freely mixes graphical and textual syntax and an algorithm has been presented to process the syntax structures. The algorithm has been validated by an implementation, but leaves room for future development. The approach is structural whilst other approaches integrate text parsing with projectional editing (*e.g.*, [4]); it may be possible to integrate both approaches. The meta-language described in this paper provides no support for error handling and will simply go wrong if the rules fail to produce a normal-form or if a local rule definition produces a tree of an unexpected type. One way to address this is to have a separate category of rules that are used for checking and error reporting. Related to this, the language does not support static checking. For example, it should be possible to detect the use of unbound identifiers and undefined functors. Some aspects of static checking should be easy to achieve, however it would also be desirable to define a type system so that the use of syntax structures can be checked before use. There is interest in the modularity and composition of languages and DSLs in particular [24, 3, 18, 19]. A key challenge to achieving engineered integration is posed by concrete syntax. By inverting the focus of attention to abstract-syntax, a projectional editor does not suffer from such problems. However, there are still significant issues to be addressed and this could be a fruitful area for future work.

References

1. Francisco Pérez Andrés, Juan de Lara, and Esther Guerra. Domain specific languages with graphical and textual views. In *Applications of Graph Transformations with Industrial Relevance*, pages 82–97. Springer, 2008.
2. Colin Atkinson and Ralph Gerbig. Harmonizing textual and graphical visualizations of domain specific models. In *Proceedings of the Second Workshop on Graphical Modeling Language Development*, pages 32–41. ACM, 2013.
3. Walter Cazzola and Ivan Speziale. Sectional domain specific languages. In *Proceedings of the 4th workshop on Domain-specific aspect languages*. ACM, 2009.
4. Lukas Diekmann and Laurence Tratt. Parsing composed grammars with language boxes. *Workshop on Scalable Language Specifications*, 2013.
5. Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science*, 253(7), 2010.
6. Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.
7. Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.
8. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Model Driven Architecture-Foundations and Applications*, pages 114–129. Springer, 2009.

9. Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language evolution in practice: The history of gmf. In *Software Language Engineering*, pages 3–22. Springer, 2010.
10. Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254. ACM, 2006.
11. Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.
12. Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.
13. Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming emf and gmf using model transformation. In *Model Driven Engineering Languages and Systems*. Springer, 2010.
14. Dimitrios S Kolovos, Louis M Rose, Richard F Paige, and Fiona AC Polack. Raising the level of abstraction in the development of gmf-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 13–19. IEEE Computer Society, 2009.
15. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, pages 297–315. Springer, 2008.
16. Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.
17. Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
18. Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. Language boxes. In *Software Language Engineering*, pages 274–293. Springer, 2010.
19. Bernhard Rumpe. Towards model and language composition. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. ACM, 2013.
20. Markus Scheidgen. Textual modelling embedded into graphical modelling. In *Model Driven Architecture–Foundations and Applications*, pages 153–168. Springer, 2008.
21. Christian Schneider. On integrating graphical and textual modeling. *Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel*, 2011.
22. Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 451–464. ACM, 2006.
23. Juha-Pekka Tolvanen and Steven Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.
24. Federico Tomassetti, Antonio Vetró, Marco Torchiano, Markus Voelter, and Bernd Kolb. A model-based approach to language integration. In *Modeling in Software Engineering (MiSE), 2013 5th International Workshop on*. IEEE, 2013.
25. Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.