

# Meta-model-based implementation of Sudoku: Eclipse vs. Visual Studio

Ingelin F. Isfeldt      Terje Gjørseter  
Andreas Prinz

Faculty of Engineering and Science  
University of Agder  
Grooseveien 36, N-4876 Grimstad, Norway  
ingelin.isfeldt@gmail.com, {terje.gjosater, andreas.prinz}@uia.no

## Abstract

By considering Sudoku as a language, where a Sudoku puzzle is an instance of the language, we are able to apply meta-model-based technologies for the implementation of Sudoku, including correctness checking of a puzzle and solving strategies. The description of Sudoku includes not only the structure of Sudoku, but also covers constraints, textual representation, graphical representation, and behaviour (transformation and execution). We have created a meta-model-based description of Sudoku and use this as a basis for a comparative implementation in Eclipse with various plug-ins, and in Visual Studio with DSL-tools.

## 1 Introduction

In this article, we describe how we have created a complete platform-independent meta-model-based language description for Sudoku and used this as a basis for implementation on two different platforms; Eclipse with various plug-ins, and Visual Studio with DSL-tools. We have chosen these two platforms since they are both popular, and also quite different in their approach to language description/modelling.

We will discuss how the implementation differs between the two platforms, with particular focus on how high abstraction level we are able to work on in the implementation; how far is the distance between the platform-independent description and the implementation. We will also keep an eye on the user-friendliness (or "developer-friendliness") of the platforms.

The platform-independent language description is further described in [3], and more details on the implementations are available in [7].

Sudoku is a very popular puzzle these days, and has been for some years now. The game of filling in the numbers from 1 to 9 into a 9x9 celled square of 3x3 celled sub-squares is fascinating people all over the world. Both smaller and larger Sudoku puzzles have become available, where letters and symbols are included in the game and also

---

*This paper was presented at the NIK-2008 conference; see <http://www.nik.no/>.*

much more advanced problems exist with colors, non-square solutions and so on. These small puzzles can be found all over the web, newspapers and magazines; and books and computer games filled with Sudoku puzzles are widely available.

## Aspects of a Programming Language

In [17], the aspects of a language description are given as *Structure*, *Constraints*, *Representation* and *Behaviour*.

**Structure** defines the constructs of a language and how they are related. This is also known as abstract syntax.

**Constraints** describe restrictions of the language structure that go beyond plain structure.

**Representation** defines how instances of the language are represented. This can be the description of a graphical and/or textual concrete language syntax.

**Behaviour** explains the semantics of the language. This can be a mapping/transformation into another language (denotational semantics), or it defines the execution of language instances (operational semantics).

A graphical model of the structure, also including constraints, is often called a *meta-model*. For a complete meta-model based language description, it is necessary to not only define the structure of the language, but also to define the representation and behaviour of the language and relate these definitions to the meta-model (see also [8]).

## Eclipse

The Eclipse platform [1] is designed from the ground up for building integrated web and application development tooling. The platform design does not provide a great deal of end user functionality by itself. The value of the platform is that it encourages rapid development of integrated features based on a plug-in model. A plug-in in Eclipse is a component that provides some type of service and/or additional features to the Eclipse workbench. The following Eclipse plug-ins were used: *Omondo EclipseUML* [16], the *Eclipse Modelling Framework* (EMF) [5], the *Object Constraint Language* tool from the *Model Development Tools* project (MDT-OCL) [9], the *Textual Editing Framework* (TEF) [18], the *Graphical Modelling Framework* (GMF) [4] and *MediniQVT* [6].

## Visual Studio

We have also used Visual Studio from Microsoft to implement Sudoku. Working with Visual Studio is in many ways very different from working with Eclipse. Visual Studio is a complete set of development tools for building ASP.NET web applications, XML web services, desktop applications, and mobile applications. Several Microsoft programming languages all use the same integrated development environment (IDE), which allows them to share tools and facilitates in the creation of mixed-language solutions [12]. The Visual Studio 2005 SDK provides a framework that can be used to extend the functionality of Visual Studio, much like plug-ins for Eclipse. The SDK includes Domain-Specific Language Tools (DSL Tools) [10] that is a component that lets the developer generate graphical designers that are customized for a specific problem.

## Organisation of the article

The rest of the article is organised as follows: Section 2 gives some background information on the game of Sudoku. The different aspects of language descriptions used to define and implement Sudoku are covered in the following sections; *structure* in Section 3, *constraints* in Section 4, *textual representation* in Section 5, *graphical representation* in Section 6, *transformation behaviour* in Section 7 and *execution behaviour* in Section 8. For each of the aspects described in sections 3 to 8, we cover the following topics: A short introduction to this aspect, how it is implemented in Eclipse and in Visual Studio, and a comparison of the two approaches. Conclusions are given in Section 9.

## 2 Sudoku

A Sudoku usually consists of a grid of 9 x 9 *cells*, divided into *rows*, *columns* and *boxes*. Some cells might already contain numbers, known as *givens* (see Figure 1). The goal of the game is to fill in the empty cells, one number in each cell, so that each row, column and box contains the numbers 1 to 9 exactly once. The puzzles come in several difficulties depending on the number and/or layout of the givens. We will use *field* as a common term for row, column and box.

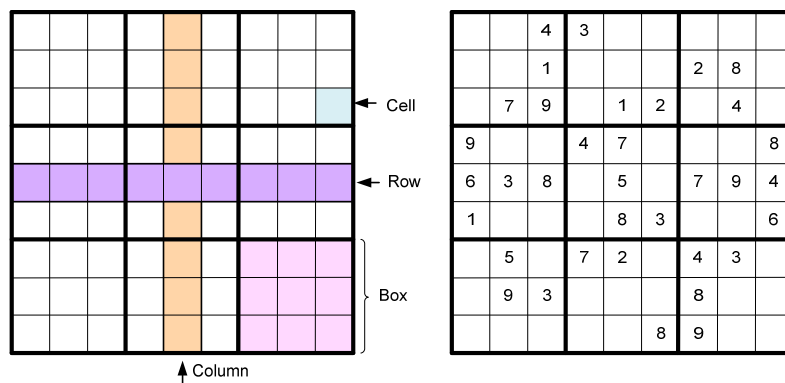


Figure 1: An empty Sudoku (left) and a Sudoku with some *givens*.

### Rules and solving strategies

There is essentially just one rule of solving Sudoku; every row, column and box must include the numbers from 1 to 9 exactly once. There are several solving strategies one can use when trying to solve a Sudoku but we will only mention a few of them here.

**Elimination:** Before trying to solve the Sudoku by using the strategies that follow, fill in all possible candidates in each cell. It is very important that when a number has been assigned to a cell, then this number is excluded as a candidate from all other cells sharing the same field. Performing this elimination is likely to narrow down the options for possible values in a cell (see [2]).

**Single candidate:** If any number is the only candidate in a cell, this number must be the correct solution for this cell.

**Hidden single candidate:** If any number is a candidate in only one cell in a field, this number must be the correct solution for this cell.

### 3 Structure

The structure of the meta-model consists of *Puzzle*, *Field*, *Row*, *Column*, *Box* and *Cell*. *Puzzle* is the root model class and it contains several *Fields*. A *Field* must have references to the *Cells* that are associated with it. *Row*, *Column* and *Box* extend *Field*. A *Cell* must have an integer *iCellValue* that contains its value, while *Puzzle* has an integer *iDimension* keeping its dimension. A *Cell* must refer to exactly one referencing *Row*, *Column* and *Box*, while a *Row*, *Column* or *Box* must refer to *iDimension* *Cells*.

#### Structure in Eclipse

The structure meta-model that was used in Eclipse was created using the Omondo EclipseUML plug-in [16]. As can be seen from Figure 2, *get* and *set* operations for *iCellValue* have been automatically added to the operations list. The reference between *Row* and *Cell* is a composition type of relationship, this means that the *Cells* are *contained* in a *Row*. Because of limitations in the tools, a *Cell* can not simultaneously be contained in more than one element, so the connection to *Column* and *Box* is a simple association, not composition.

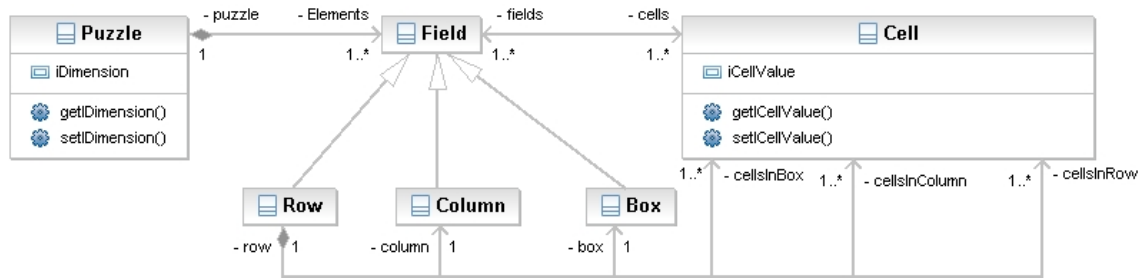


Figure 2: A structure meta-model for Eclipse.

#### Structure in Visual Studio

Visual Studio has its own designer for setting up the DSL structure, called *DomainModel*. A *DomainModel* has *DomainClasses* and *DomainRelationships* similar to classes and relationships in UML, and is therefore similar to a class diagram. Figure 3 displays the structure diagram from Visual Studio. The lines that are visible from *Row* and *Cell* *DomainClasses* are relations to these classes' graphical elements and do not influence the structure of the model. The relationships look a bit different from what we are used to from UML. The *DomainRelationship* *PuzzleHasElements* is an embedding relationship, much like the UML composition relationship. Attributes can be added to the *DomainRelationship*. This is automatically added to the *DomainModel*. The reference between *Row* and *Cell* is also different as it is an embedding type of relationship. This is done to better fit the development in Visual Studio with DSL Tools.

#### Comparison

It was very easy and straightforward to create the structure in both Visual Studio and Eclipse. The two implementations share a quite high level of abstraction. The Visual Studio differs in some way from the Eclipse-based meta-model in the sense that *DomainRelationships* are created automatically.

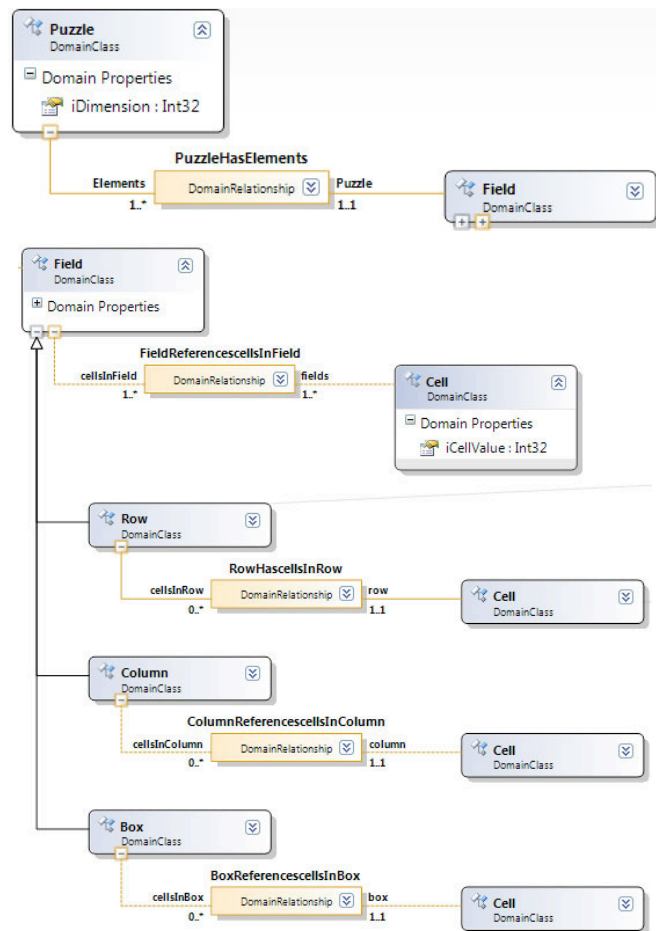


Figure 3: A part of the structure meta-model from Visual Studio.

## 4 Constraints

Constraints are a valuable help for model validation. The structure defined in Section 3 does not provide all necessary limitations that this project demands. For example, we need to make sure that a *Row* has exactly *iDimension* cells.

### Constraints in Eclipse

The EMF [5] and MDT OCL [9] make it rather easy to implement OCL constraints in a Java application. It enables you to set the context for your queries and use OCL statements to query your model.

The following code example shows an example constraint code for checking that a *Row* has exactly *iDimension* Cells.

```
context Puzzle inv PuzzleRow:
self.Elements->select(f : Field | f.oclIsTypeOf(Row))
-> size()=iDimension
```

MDT OCL demands a large amount of extra code in addition to the OCL statements. Thus knowing OCL is not enough to actually implement the constraints. For the line of OCL code above, approximately 10 lines of helper code is needed, for example code for setting the context of the query and the query itself. The constraints are checked for

validity by a simple if-else-branch, providing an error message if the query evaluates to false. There is no way to specify the severity of constraint violation, e.g. separating errors, warnings and messages.

## Constraints in Visual Studio

In Visual Studio, constraints are added as validation code in your preferred VS programming language. In this project the constraint code is written in C#. To avoid interfering with the generated code, the validation methods are written in separate files that define partial classes [13] where the constraint code is put. This also prevents custom code from being deleted if code is regenerated. We created three different code files for constraints, separating the *Puzzle*, *Row/Field* and *Cell* constraints. DSLTools allows returning both messages (informative only), warnings and error messages as well as assigning an error message number to each error. One can also specify when a constraint shall be evaluated.

All constraint violations result in error messages provided in the code. The Visual Studio constraints solution differs from the platform independent constraints solution as it can not use OCL or logic constraints, but relies on C# code.

## Comparison

The biggest difference between implementing constraints in Eclipse and Visual Studio is the code language. For the Eclipse based solution, OCL is the natural choice for constraints. Thus we are allowed to work on a relatively high level of abstraction when it comes to defining the constraints, compared to Visual Studio. Visual Studio does not support OCL but demands some Microsoft programming language, thus the constraints in Visual Studio have been implemented using C#. On the other hand, Visual Studio has a more abstract approach to embedding the constraints in the project than Eclipse with MDT OCL, allowing the constraint code to be put in separate files.

## 5 Textual Representation

For the textual representation of Sudoku, we want to create a customised textual editor. The editor should take care of error handling and creating the correct internal structure. The textual editor for Sudoku should be simple and straightforward. It should let the user create a Sudoku puzzle, either a complete (solved) puzzle or a ready-to-solve puzzle with only *givens* filled out, by filling Rows with Cells simply by writing text in the form of lists of numbers. For defining editors and parsers for a textual representation, grammars are commonly used.

### Textual Representation in Eclipse

With TEF [18] it is relatively easy to construct a textual editor by defining a grammar referring to a meta-model. You are then free to write your grammar specifying the necessary elements and creating your own syntax based on the provided model.

To create a complete (solved) Puzzle with the TEF-based editor, one can write:

```
Puzzle (9)=  
Row (6,7,3,8,2,1,9,4,5),  
Row (5,1,4,3,9,6,2,7,8),  
Row (9,2,8,5,7,4,3,1,6),
```

Row (1,4,6,2,8,9,5,3,7),  
Row (7,8,9,6,3,5,4,2,1),  
Row (2,3,5,1,4,7,8,6,9),  
Row (3,9,2,7,1,8,6,5,4),  
Row (4,6,1,9,5,3,7,8,2),  
Row (8,5,7,4,6,2,1,9,3)

This creates a new Puzzle with *iDimension* 9 containing nine *Rows* all containing 9 *Cells* with an *iCellValue*. We had to use a trick (overriding the default constraint check method of TEF), in order to get a correct model instance from the editor where *Cells* are also connected to *Columns* and *Boxes*.

## Textual Representation in Visual Studio

By using Visual Studio 2005 with DSL Tools, it is not possible to create textual DSLs, and it does not seem to be something that Microsoft plans to implement in the future as DSL Tools is meant for creating graphical DSLs.

## Comparison

With Eclipse and TEF (Textual Editing Framework) [18], it is relatively easy to create a text editor with features such as syntax highlighting, code completion, intelligent navigation, and visualization of errors and warnings. Visual Studio with DSL Tools does not support the creation of textual editors at all.

## 6 Graphical Representation

For the graphical representation we want editors that let a user edit a graphically presentable Sudoku. Ideally when the user starts a new Sudoku editor, she should be presented with a ready-to-use Sudoku that contains the correct number of Rows containing Cells, Columns and Boxes. All references must be handled automatically. Only Row and Cell need graphical elements, the rest should be handled in the background.

### Graphical Representation in Eclipse

GMF (Graphical Modeling Framework) [4] allows the creation of graphical elements and the mapping of these graphical elements to model elements. This lets the developer create graphical editors for their models in a relatively simple way. GMF provides wizards to lead the developer through large parts of the development including code generation. Custom code can be introduced by overriding the generated methods. The GMF takes an Ecore model as input and combines this model with graphic- and tool definitions to create a mapping definition and then a generator model that generates an editor/diagram plug-in based on the provided model and definitions.

The GMF solution allows to specify for example element location and size. The graphic definition is held in the *\*.gmfgraph* file. By adding graphical elements and editing properties of these elements in this file, one can customize the graphical elements for all model elements. Shape, color, image, Sudoku meta-model size and layout are among the possible properties for these elements as well as lines for displaying relationships (not necessary in this project). In addition, one can create tools for creating new elements by drag-and-drop and mappings between tools, graphical elements and model elements. Cell values are edited simply by selecting a cell and entering the new value.

To provide the users of the editor with a ready-to use Sudoku when starting a new diagram, the elements must be created manually by adding custom model code into the generated code. GMF generates a lot of code, so it is not always easy to find out where to put your custom code in order for it to work properly. The elements and all necessary relationships are created using the EMF.

## Graphical Representation in Visual Studio

Visual Studio is built specifically for creating graphical designers for domain-specific languages [11]. The Sudoku designer lets the user edit her own Puzzle by assigning values to cells. Each Row and Cell is represented by graphical elements, a Row containing 9 Cells.

It is relatively easy to create these graphical elements in Visual Studio. There are several starting templates, which get you started with some elements that you can change to fit your need and of course add new elements. Using the same model as in the structure part, you can create a diagram element, e.g. shapes for each class for which you need graphical elements. These shapes are easily mapped to the corresponding class in the model. In the properties view you can change the appearance of the shape, like color, line thickness, geometry, and size. For the graphical editor, a *Cell* and a *Row* have the same height, but the *Row* width is 9 x a *Cells* width.

Custom rules can be added to the DSL to fire on specific events. We have created custom rules for the events when a *Row*, *Column*, *Box* or *Cell* is added to the model.

When a new Sudoku is created, it is already filled with 9 *Rows* containing 9 *Cells* that are visible to the user. In addition to this, there are also 9 *Boxes* and *Columns* that have the necessary references to *Cells*.

## Comparison

Graphical representation is well supported in both Visual Studio with DSL Tools and in Eclipse with GMF. Both tools allows the developer to create advanced graphical editors with relative ease.

## 7 Transformation behaviour

In model-to-model transformations, a model conforming to a given meta-model is converted into another model conforming to a given meta-model that may be the same as the first, or a different one. In our case, the meta-model is the same. We will use in-place transformations, meaning that the source and target models are also the same.

The following transformations can be performed on a Sudoku without altering the logic:

- Permutations of *Rows* and *Columns* within *Blocks*.
- Permutations of *Block* rows and *Block* columns.
- Permutations of the symbols used in the board.
- Transposing the Sudoku (changing *Rows* to *Columns* and vice versa).

A *Block* refers to a row (or column) of *Boxes*, e.g. *Rows* 1, 2 and 3 form a block.

One of these transformations have been attempted implemented: Permutations of the symbols used in the board. The symbols are in this project the numbers from 1 to 9. By

permutation we can arrange the Sudoku in such a way that the *Cells* in the first *Row* are ordered ascending from 1 to 9. In order to achieve this, the *iCellValue*s from the first *Row* must be retrieved, and then used to switch all the *iCellValue*s in the *Puzzle*. E.g. if the first *Cell* in the first *Row* has *iCellValue* = 6, then all *Cells* where *iCellValue* = 6 must be changed to 1.

## Transformation in Eclipse

QVT (Queries/Views/Transformations) [14] is a standard for model-to-model transformations defined by the Object Management Group (OMG). QVT integrates the OCL 2.0 standard [15] and also extends it to imperative OCL. It defines three Domain Specific Languages named Relations, Core and Operational Mappings, that are organized in a layered architecture. Both the Core and the Relations language are declarative languages (at different levels of abstractions), with a mapping between them. The Relations language has both graphical and textual concrete syntax. The Operational Mappings language is an imperative language that extends the Core and Relations languages.

One of the available QVT plug-ins for Eclipse is mediniQVT [6]. It supports the QVT Relations language, and allows execution of QVT transformations expressed in the textual concrete syntax of the Relations language.

To use medini QVT for model-to-model transformations one must provide meta-models for the models to be transformed. In this case the Sudoku meta-model will act as both source and target meta-model.

To sort the first *Row* by permuting *Cell* values, all *Cell* values of the first *Row* must be retrieved. The *Cell* values throughout the *Puzzle* must be changed to the same value as the same value's original position in the first *Row*. This means that if the *iCellValue* in the first position of the first *Row* was 6, then all 6 in the Sudoku must be changed to 1. If the value in the second *Cell* was 3, then all 3's must be set to 2 and so on. This turns out to be problematic as there might already be e.g. a number of 2 in the *Puzzle*. Then, if the value in *Cell* 3 is 2, all 2's must be changed to 3. However there are really two kinds of 2, the kind that is already correct and the ones that are not. Therefore some temporary values are used for the values that are being changed to avoid this problem.

## Transformation in Visual Studio

The only built-in transformation tool for DSL Tools is the text templates. In DSL Tools a text template is a file that can contain both text blocks and control logic. When a text template is transformed, the control logic combines the text blocks with the data in the model(s) in question, to produce some output file. This file can be a code file like C#, Java, HTML or just pure text controlled by the file extension you decide and the text itself. The other option is to perform transformations using C# code. As we are interested in a model-to-model transformation, the second option is our choice.

To sort the first *Row*, all *iCellValue*s from this *Row* are collected. If the *iCellValue* of the first *Cell* has *iCellValue* 9, then all *iCellValue*s in the *Puzzle* that have this value will get the value 1 instead. By performing this change for all *Cells* in the first *Row*, changing values, the result will be that the first *Row* has the values from 1 to 9 and as all values are changed accordingly, thus the Sudoku is still valid. This solution contains a lot of extra code besides the actual sorting. An excerpt of the sorting code is shown below:

```
foreach (Field field in puzzle.Elements){
    if (field is Row){
```

```

countrow++;
if (countrow == 1){
    foreach (Cell c in field.cellsInField){
        //Retrieve all iCellValues from row 1
        countcell++;
        if (countcell == 1)
            {iCellValue1 = c.iCellValue;}
        if (countcell == 2)
            {iCellValue2 = c.iCellValue;}
        //and so on for 3-9
    }
}
foreach (Cell c in field.cellsInField){
    using (Transaction t =
        store.TransactionManager.BeginTransaction("")){
        //Set the correct new iCellValues
        if (c.iCellValue == iCellValue1)
            c.iCellValue = 1;
        else if (c.iCellValue == iCellValue2)
            c.iCellValue = 2;
        //and so on for 3-9
        t.Commit();
    }
}
}
}

```

All *iCellValues* from the first Row are stored in int *iCellValueX* where *X* refers to the *iCellValues* original position in Row 1. Then for all the *Cells* in each Row the new *iCellValue* is set so that if *iCellValue = iCellValueX*, then *iCellValue = X*.

## Comparison

In Visual Studio, C# was used to define the transformations. In Eclipse, we expressed the transformations using mediniQVT. QVT may theoretically allow the developer to operate on a slightly higher level of abstraction than C#, but the implementation met with some difficulties since QVT is quite difficult to master for first-time users.

## 8 Execution behaviour

The execution part in this project consists of some solving strategies of Sudoku that will be run when the model is executed. The strategies that we have attempted to implement are: *Elimination*, *Single Cell candidate* and *Hidden single cell candidate*.

To execute these, we need a *runtime environment*, i.e. a runtime state space. This can be expressed as an extension of the meta-model. In order to perform the solving strategies, we need a new attribute for *Cell*, preferably a List<int>. This list will be named *iPossibleCellValues* and we will use it to keep track of the values that are available in a *Cell* at any given time during the solving process.

We also need to define the initial state of the system; this can be described as an object diagram, or simply with constraints. Finally we need to define the state transitions of the system.

A *Cell* is solved when its List *iPossibleCellValues* only contains one value. If a *Cell* only has one possible value, this must be the correct value for this *Cell* and its *iCellValue* is set to this value. When execution is initiated, values are deleted from the

*iPossibleCellValues* by the rules of the implemented solving strategies. A *Puzzle* is solved when all *Cells* are filled (not empty) and the *Puzzle* is valid.

## Execution in Eclipse

We have attempted to create the execution part of the Eclipse based solution using mediniQVT. We consider mediniQVT to be suitable since execution of the model is really a set of transitions between states, and therefore not too different from transformations.

## Execution in Visual Studio

In Visual Studio, execution is as all other aspects implemented in your preferred language for VS, in our case C#. We need a List type *DomainProperty* in the *Cell DomainClass*. This type is strangely enough not supported in the DSL Tools structure; thus we had to create the type we needed ourselves. In addition, a label displays how many *Cells* have been solved and what solving strategies are used.

## Comparison

As for transformations, C# was used to define the execution behaviour in Visual Studio, and mediniQVT was used with Eclipse, so the comments in the comparison part of Section 7 are also valid here.

## 9 Conclusions

Eclipse and Visual Studio are definitely very different environments to work with. One advantage of using Visual Studio is that everything is available to the user/developer in one single solution. For the Sudoku implementation: structure, constraints, graphical editor, execution and transformation are all available in one single project/solution. Using Eclipse, most of these features depend on some plug-in, thus to cover all these aspects several plug-ins must be installed. Each plug-in might require a specific kind of project or setup, and one project actually ends up with several projects: the main project, the designer project, the editor project and so on.

However, the idea and purpose of Eclipse is to use it with plug-ins, and this leaves the developers with the privilege (and the problem) of choosing the preferred plug-ins that fit their project best. For example when several tools for creating textual editors are available, one can choose to work with the one that fits the current need best. Using Visual Studio, this is not an option. The amount of plug-ins available for Eclipse is increasing every day, making even more functionality available to developers. We also noted that both tools had some stability issues.

To sum up our experiences, Visual Studio is good on integration, documentation and ease of use, while Eclipse in general allows the developer to operate on a higher level of abstraction and has a good selection of plug-ins.

## References

- [1] d'Anjou, J., S. Fairbrother, D. Kehn, J. Kellermann and P. McCarthy, "The Java Developer's Guide to Eclipse," Addison-Wesley, (2004).
- [2] Delahaye, J.-P., *The science behind sudoku*, Scientific American (June 2006), pp. 80–87.

- [3] Gjørseter, T., I. F. Isfeldt and A. Prinz, *A language description for sudoku*, to appear in the proceedings of the Software Language Engineering 2008 workshop (in Springer's Lecture Notes in Computer Science series).
- [4] GMF developers, "Eclipse Graphical Modeling Framework," See also <http://www.eclipse.org/gmf> (accessed 2008-06-25).
- [5] Griffin, C., *Using EMF*, Technical report, IBM: Eclipse Corner Article (2003), also available at <http://www.eclipse.org/articles/Article-Using-EMF/using-emf.html> (accessed 2008-10-14).
- [6] ikv++ developers, "Medini QVT," ikv++ technologies ag, see also <http://projects.ikv.de/qvt> (accessed 2008-06-25).
- [7] Isfeldt, I. F., "A metamodel for Sudoku," Master's thesis, University of Agder, Also available at <http://student.grm.hia.no/master/ikt07/ikt590/g01/> (accessed 2008-10-15) (2008).
- [8] Kleppe, A., *A language is more than a metamodel*, in: *ATEM 2007 workshop*, (2007), available at <http://megaplanet.org/atem2007/ATEM2007-18.pdf> (Accessed 2008-10-14).
- [9] MDT-OCL developers, "MDT-OCL," see also <http://www.eclipse.org/modeling/mdt/?project=ocl> (accessed 2008-06-25).
- [10] Microsoft, "DSL Tools," see also [http://msdn.microsoft.com/en-us/library/bb126235\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/bb126235(VS.80).aspx) (accessed 2008-06-25).
- [11] Microsoft, "Getting Started with Domain-Specific Languages," see also <http://msdn2.microsoft.com/en-us/library/bb126278.aspx> (accessed 2008-06-25).
- [12] Microsoft, "Introducing Visual Studio," see also [http://msdn2.microsoft.com/en-us/library/fx6bk1f4\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/fx6bk1f4(VS.80).aspx) (accessed 2008-06-25).
- [13] Microsoft, "Visual Studio SDK," see also [http://msdn2.microsoft.com/en-us/library/bb166441\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/bb166441(VS.80).aspx) (accessed 2008-06-25).
- [14] OMG, *MOF 2.0 Query/View/Transformation Specification Final Adopted Specification ptc/05-11-01*, OMG document, Object Management Group (2005), also available at <http://www.omg.org/docs/ptc/05-11-01.pdf> (Accessed 2008-06-25).
- [15] OMG, "OCL 2.0 Specification," Object Management Group, (2005), ptc/2005-06-06.
- [16] Omondo developers, "OmondoUML documentation," see also <http://www.eclipsedownload.com/documentation.html> (accessed 2008-06-25).
- [17] Prinz, A., M. Scheidgen and M. S. Tveit, *A Model-based Standard for SDL*, in: *SDL 2007: Design for Dependable Systems*, Lecture Notes in Computer Science 4745 (2007), pp. 1–18.
- [18] Scheidgen, M., "Textual Editing Framework," see also <http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/documentation.html> (accessed 2008-06-25).