Electronic Communications of the EASST
Volume 8 (2008)

Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007)

Novel Techniques For Model-Code Synchronization

László Angyal, László Lengyel, and Hassan Charaf

14 Pages

Guest Editors: Tom Mens, Ellen Van Paesschen, Kim Mens, Maja D'Hondt
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Novel Techniques For Model-Code Synchronization

László Angyal, László Lengyel, and Hassan Charaf

Budapest University of Technology and Economics
{angyal, lengyel, hassan}@aut.bme.hu

Abstract: The orientation of the current software development practice requires efficient model-based iterative solutions. The high costs of maintenance and evolution during the life cycle of the software can be reduced by using tool-aided iterative development. This paper presents how model-based iterative software development can be supported through efficient model-code change propagation. The presented approach facilitates bi-directional synchronization between the modified source code and the refined initial models. The backgrounds of the synchronization technique are three-way abstract syntax tree (AST) differencing and merging. The AST-based solution enables syntactically correct merge operations. OMG's Model-Driven Architecture describes a proposal for platform-specific model creation and source code generation. We extend this vision with the synchronization feature to assist the iterative development. Furthermore, a case study is also provided.

Keywords: MDA, MIC, Model-Based Iterative Development, Three-Way AST Differencing, AST Merging.

1 Introduction

The current model-based development approaches emphasize the use of models at all stages of software development. Models are used to describe all artifacts of the system, i.e., user interfaces, interactions, and properties of all the components that comprise the system.

The MDA [1] approach separates the platform-independent application specification from the platform-dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM) and one or more platform-specific models (PSM) on those platforms that the application developer decides to support. The platform-independent artifacts are mainly Unified Modeling Language (UML) [2] and other software models containing enough specification to generate the platform-dependent artifacts automatically by model compilers. UML is a widespread general purpose modeling language that is too general to support efficient source code generation. Closely focusing on a specific problem domain, domain-specific languages (DSL) cover a narrow domain-specific area in a high abstraction level. Therefore, greater part of the source code generation can be covered by domain-specific model processors.

Model-Integrated Computing (MIC) [3] provides a framework for software production using both metamodeling environments and model interpreters. The domain-specific concepts of MIC are similar to MDA. MIC supports the flexible creation of modeling environments by metamodeling basics, and helps tracking the changes of the models. Using MIC technique the domain-specific models can be converted into another format such as executable code.

One of the most important challenges of model-based development is round-trip engineering [4]. Almost every existing modeling tool supports generating the code skeletons, and synchronization of the skeleton with the models. However, it is insufficient in most cases. Usually, the generated source code does not contain enough logic to be used effectively without further manual programming. The current reverse engineering tools mostly focus on the inter-procedural calls or pattern recognition, and there is lack of tool support for the statement-level handling of method bodies, but it is also important. The method bodies cannot be supported without the comprehension of their logic. Textual-based solutions are unable to extract enough information, therefore an AST-based approach is required.

In order to support model-driven iterative development, software models and the source code should be kept synchronized. This work provides a three-way AST difference analysis and merge-based software model-code synchronization solution. The remainder of the paper is organized as follows. The next section discusses the motivation. Section 3 provides background information and related work, Section 4 describes our model-code synchronization approach and a concrete platform-specific solution with a case study. Finally, conclusions and future work are elaborated.

2 Motivation

Our motivation is to support automatic, iterative application generation and synchronization with the same platform-independent model set for different platforms, e.g. for different mobile phone platforms (Fig. 1).

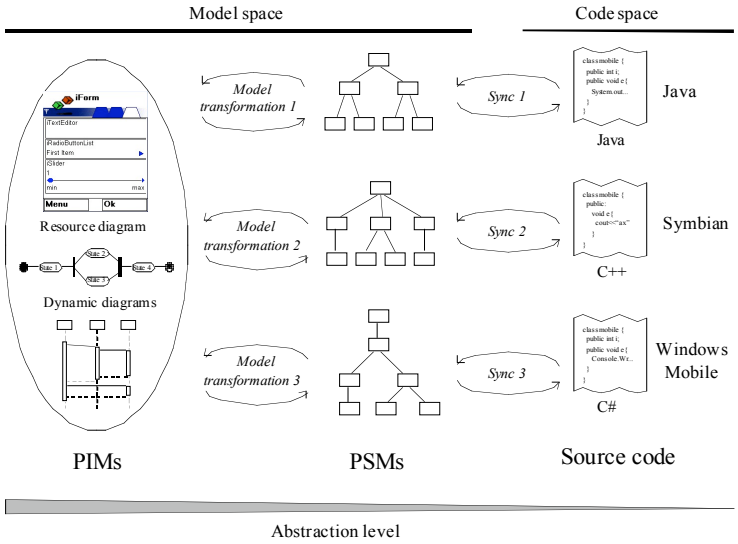

Fig. 1. Motivation principles

The model space consists of the platform-independent models (PIMs) and the platform-specific models (PSMs). The PIMs are composed of static resource and other dynamic models (Fig. 1), which describe the user interface and the behaviour of the application. A PSM represents a concrete implementation of the PIMs in a selected platform. Several development platforms are available for mobile phones e.g. Symbian [5], Java Micro Edition [6], Windows Mobile [7].

Generally, any PSMs can be produced from the PIM using graph rewriting-based [8] model transformations. PSMs are constructed to facilitate source code generation. Visual Modeling and Transformation System (VMTS) [9] is an n-layer metamodeling environment which supports visual editing of models according to their metamodels, and allows specifying Object Constraint Language (OCL) constraints in model transformation rules. Both the models and the transformation rules are represented internally as directed, labeled graphs stored in the VMTS's database. Also, VMTS is a model transformation system, which transforms models using graph rewriting techniques and supports code generation from models. VMTS facilitates an efficient and simple way to define model transformations visually.

The relations between the AST elements can be considered the metamodel of the language and its instance model as a concrete AST of a source code that is written in that language. According to [10], an AST metamodel is powerful, because the manipulation of programs became universal. The regeneration of the code from an AST model is possible via model traversal and parsed source code can be stored in that model. Thus, choosing an AST model as PSM can be advantageous.

Both PIMs and the source files generated from the PSMs can be modified by the developers. The model and the source code are modified independently of each other over the time. To solve the synchronization problem, this paper introduces how VMTS supports model-based iterative software development.

In order to avoid the error-prone manual change propagation, the synchronization feature between these states is essential. The bi-directional non-destructive model-code change propagation enables iterative and incremental software development (IISD). The non-destructive way means that the previously committed changes in both sides will be saved. The whole development process can be split into iterations instead of sequential phases. After each iteration, developers have executable software to use and to draw conclusions from it, and to incorporate the observations into the next iteration of the project.

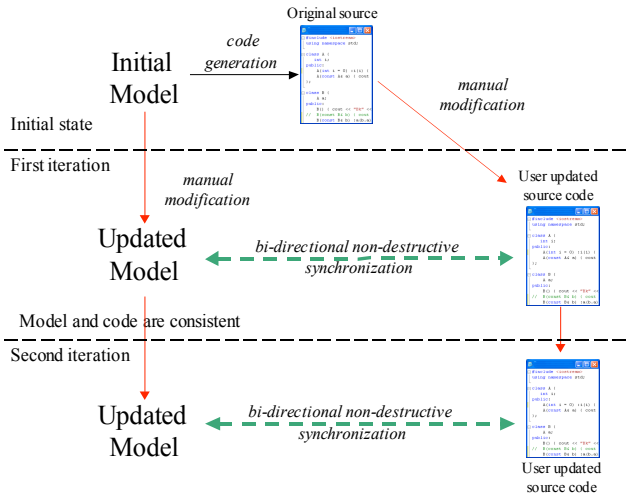

Fig. 2. Iterative model-based development

Fig. 2 depicts a typical scenario of IISD. In the beginning of the first iteration, we generate code from a model. The development affects both the model and the code. At the end of each

iteration, before the start of the next one, we should bring the model and the code into a consistent state. In model-driven iterative development the synchronization between PSMs and source code should be supported. Due to the incremental synchronization, the model is always up to date, developers can choose between further source code modifications and additional model refinements. This paper describes a possible solution to model-based IISD.

3 Background and Related Work

Considerable research was conducted in the subject of source code differencing and merging in the past 30 years. Software development projects are typically based on teamwork, where the team mates work on many kinds of common files. The methods of the source code comparing and merging were and will always be an actual problem. The comparison of two files as sequences is originated in string operations, like finding the longest common subsequence (LCS) [11] or the longest common substring [12]. The two algorithms cover different matching approaches. The traditional file comparing tools usually use modified versions of the longest common subsequence searching algorithms.

The difference analysis of two strings (S_1, S_2) or trees (T_1, T_2) produces an edit script [11]. An edit script contains the operations needed to transform S_1 into S_2 or T_1 into T_2. The analysis first matches the common parts then generates the script that contains as few operations as possible, such as a minimal list. For example, the popular UNIX *diff* utility [13] finds the longest common subsequence of the lines of the two files, then creates a sequence of insert and delete operations.

Diff works well on general text files, but in several cases when differencing source code files, *diff* can produce useless edit scripts, because *diff* does not know the specific features of the programming languages. To compare two pieces of source code correctly, the algorithm must take the grammar of the language into consideration. These algorithms work on the parse trees of the files and use different approaches. Hierarchical structures such as trees cannot be handled as such simple sequences.

[14] provides an algorithm for change detection in hierarchically structured information that has linear time complexity. This approach is supposed to be used on ordered trees with large amount of data. *FastMatch* and *EditScript* algorithms are provided for the two sub-problems of the change detection problem. The *FastMatch* uses some heuristic solutions e.g. equality comparison of two labeled nodes containing a value for computing the unique maximal matching. The algorithm *EditScript* supports the insert/delete/update and the complementary move operations. The refined versions of the algorithms have been adopted in our proof-of-concept implementation that is mentioned later. The improved versions [15] of these algorithms work on unordered trees and allow copy/glue operations, as well.

Merge performs reconcile of changes in some revisions of a software artifact. Merge firstly applies the comparison of the files in order to detect the incorporated changes to be propagated. Distinction can be made between the numbers of artifacts e.g. files processed by the merge. The two-way approach works on two revisions (A_1 and A_2) of the same software artifact. Typically this is error-prone and requires human intervention and verification, because this is unable to unambiguously detect the modifications. Therefore, this technique is rarely used in practice. According to the three-way merge approach, the common ancestor (A_0) of two modified artifacts (A_1 and A_2) is used, to detect and resolve more conflicts, e.g. two-way merge is unable to distinguish insert and delete operations. The change propagation is

performed after the difference analysis between A_1 and A_2, which takes account of A_0. This approach can unambiguously detect the changes, therefore, this is the most reliable in practice. The three-way comparison with automatic conflict resolution allows automated merge. However, in several cases, human validation of the conflict management is still required to obtain source code that compiles. The best-known tool that uses three-way merge approach is *diff3* [16].

Further information can be found about other software merging techniques in [17].
Merge tools that work on parse trees have to output the reconciled source code from the tree. Pretty-printing [18] is a technique that makes an internal structure human readable. It traverses the AST and visits every node while printing the output. It facilitates the source code generation from an AST. The tokens belonging to the nodes are printed in order. The inserted white spaces and blank lines make the output pretty. It is also known as code beautifier, which makes a code well-indented.

[19] addresses the conflicts during three-way merge of hierarchically structured documents. The possible eleven combinations of operations on the pairs of child nodes during merge are enumerated.

3DM [20] and *DeltaXML* [21] are tools for performing differencing and three-way merging of XML files. *3DM* applies heuristic tree matching between trees. This performs node similarity measurements i.e. content similarity and child list similarity when no exact matches exist. The tree matching of *DeltaXML* uses an optimized longest common subsequence (LCS) algorithm, which matches the nodes at each level on the trees, and then the differences are reconciled.

Syntactic merge techniques using trees or graphs as underlying data structure. [22] proposes a program merging system that consist of a syntax-based comparator, a synchronous pretty-printer, and a merging editor. The synchronous print simultaneously traverses the trees that is previously matched by the comparator, and produces an intermediate merged file that can be edited further with the merging editor, the differences are coloured in the editor to the user can verify the merge.

There is an other interesting approach in [23], where a graph-based software merging using a category theory approach is presented. The proposed approach does not depend on any specific programming language. The programs are represented as graphs and the approach uses categorical formalisms.

The novelty of this work is the combination of three-way tree differencing, edit script controlled tree patching (merge) and model-driven development approach to achieve the synchronization of the model and code spaces. Most of the tools and cutting edge researches in the field of synchronization focus on differencing and merging between the same kinds of artifacts (mostly XML data files and source code).

4 Contributions

First of all, in this section, a novel PSM-code synchronization approach is elaborated, that can perform a syntactically correct three-way merge. That is followed by a concrete platform-dependent solution of VMTS with a case study. The elaboration of such model transformations that can synchronize the PIM-PSM is not in the scope of this paper.

4.1 The General Synchronization Method

The following method is based upon the assumption that the structure of the PSM and the AST built by the parser are conformable or at least very similar. The synchronization method depicted in Fig. 3 is based on AST differencing. The files containing the source code (S_0, S_1) are parsed into ASTs. In the model space the PSM (M_1) describes the concrete implementation in a concrete platform. The PIMs, omitted from Fig. 3, and PSMs are stored in the model database. In our case the PSM is an AST model that can contain directly the AST representation of the platform-specific code. This makes easier to match and then merge the two different representations.

Obviously, the complex AST model (PSM), which is based on model containment hierarchy, is not editable directly by the developer. Due to the large number of nodes and the absence of the layout, it is nearly impossible to visualize it. PSMs that contain AST representation are modified indirectly via model transformations or model-code synchronization. Our solution uses three-way approach starting from the persistently stored artifacts (S_0, S_1, M_1). The four phases of our synchronization method are (i) the auxiliary tree building phase, (ii) the edit script generation phase applying tree differencing, (iii) edit script transformation phase, and finally (iv) the three-way tree merge phase. The last synchronized AST model (M_0) is logically equal to the common ancestor (S_0) that reflects the last synchronization state. It is important that both of them contain the same implementation state, but in different representations.

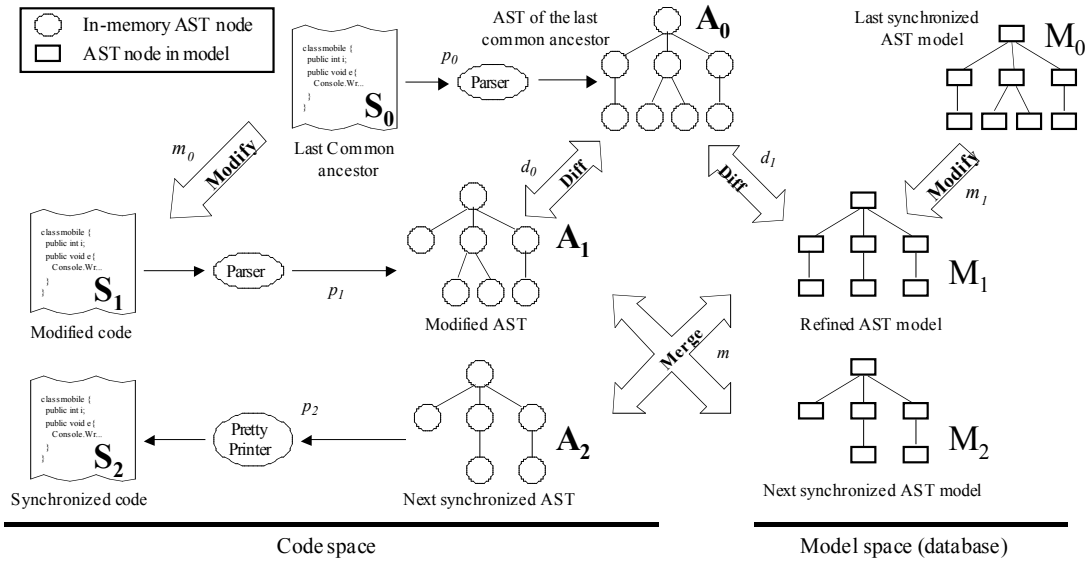

Fig. 3. Block diagram of the model-code synchronization

Both the model and the code can be edited independently and the synchronization means change propagation between them. The resulting artifacts of the merge are S_2, and M_2. The modified code (S_1) and the refined model (M_1) will be overwritten by the synchronized ones (S_2 and M_2). The others are temporary memory objects (A_0, A_1, A_2).

The difference analysis (d_0, d_1) between the two modified software artifacts (S_1 and M_1) and the common ancestor (S_0) finds and identifies the applied edit operations. Since the underlying

data structures are trees, these are atomic tree node operations. The source files (S_0, S_1) are parsed (p_0, p_1) into abstract syntax trees (A_0, A_1), and the PSM (M_1) is already an AST structure. The differencing (d_0, d_1) produces the merge instructions, which are used during the merge phase (m). Then using the pretty-printing technique (p_2) source code (S_2) can be generated i.e. the in-memory AST is serialized. The model (M_2) is updated directly, no other operations are needed. In order to provide further details related to the presented method, Fig. 4 illustrates a detailed version of the synchronization process depicted in Fig. 3. Trees denoted by A_0, A_1, A_2 are in the code space, while M_1, M_2 are in the model space according to Fig 2, but T_0, T_1 are temporary trees for helping the difference analysis.

The first phase of the synchronization builds two auxiliary trees (T_0, T_1) from the two different representations of the implementation using the visitor pattern [24] and model traversal. In the second phase, the modifications made on the source code and the model refinements are identified, and edit scripts (ε_1 and ε_2) are produced. The differencing algorithm (d_0, d_1) works on the auxiliary trees, instead of comparing directly the AST in the memory and the PSM in the database. The homogeneous auxiliary tree representation is designed to fit the tree matching algorithm. The heterogeneous structure of an AST is difficult to traverse using a general algorithm, because the different types of nodes are not comparable. The advantage of this method is that it hides the concrete way of storage of the AST and model traversals: the differencing (d_0, d_1) and the edit script processing algorithms (a_0, a_1) are the same for arbitrary programming languages.

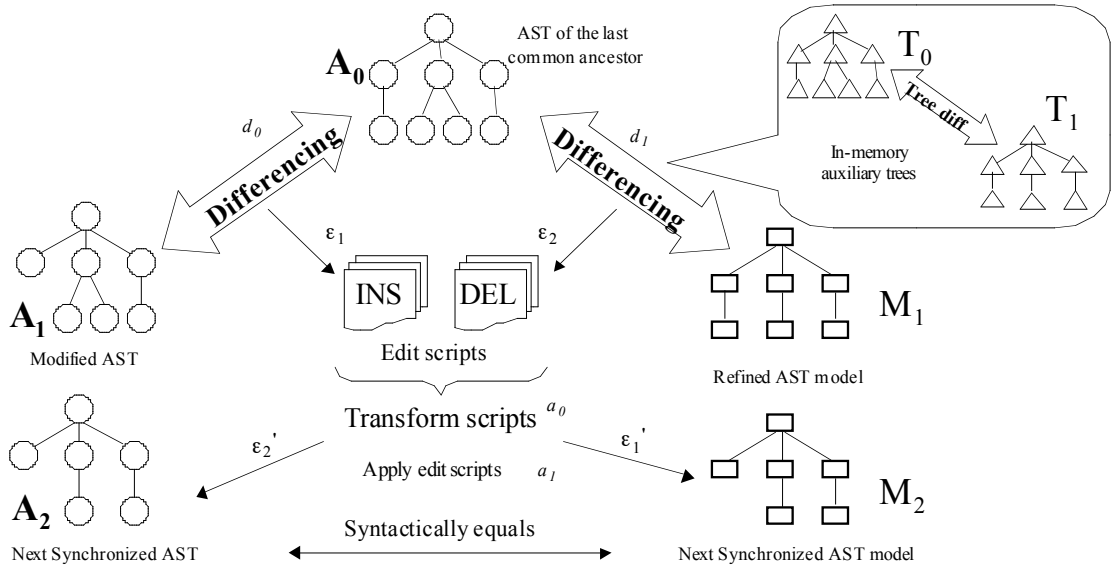

Fig. 4. The details of synchronization

Supported edit operations in our method are the following: (i) moving a subtree, (ii) insert, (iii) delete, and (iv) update a node, because they are the typical AST modification operations. Since the model representation is in a database, therefore, recognizing subtree movements can keep the number of executed row inserts/deletes low during the synchronization. A node update occurs e.g. when the name of a method or the return type is changed. Reordering two statements in a method body causes a subtree move operation. Each operation refers to exactly

one tree node that is affected by the operation, which can be a root of a subtree, its parent node and its index, as well.

A merged edit script should be created from the two already existing scripts ($\varepsilon_m=\varepsilon_1+\varepsilon_2$). This script could be executed later on the common ancestor (A_0) and the result is a synchronized AST that contains all modifications. In our case there is no representation of A_0 in each space, since it is stored only as a file (S_0). Therefore, instead of using ε_m, we produce two transformed edit scripts (ε_1' and ε_2') in order to execute them on the opposite side (A_1 or M_1) to create the synchronized artifacts denoted by A_2 and M_2. This does not require storing the ancestors in both spaces, only a file in the code space.

While transforming the edit scripts, it is important that the operations from different edit scripts can affect on each other. Since operations store positions, and inserting/deleting a node shifts the indices of the children of a parent node, positions should be maintained to be suitable for the application on the opposite AST. The conversion of the edit scripts (i) updates the positions of the contained operations if necessary, (ii) removes the overlapping delete operations and (iii) resolves the conflict between two edit operations. The conflict resolution needs labeling one side as master side. Since from the model-based development point of view the model representation is the more relevant, by default, models are the primary artifacts.

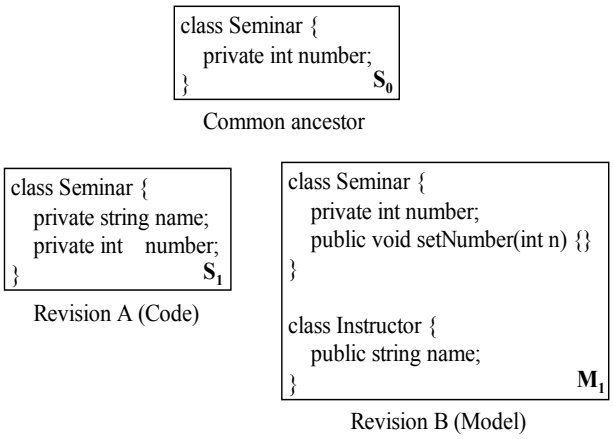

Fig. 5. A small example

To illustrate the edit script correction phase, we introduce a small example in Fig. 5. Revision B represents the rendered view of the AST model. The matching finds the correspondence between the two pairs of trees. The edit scripts, derived from the non-matched nodes, are listed in Table 1 and Table 2 (the affected nodes got informative names).

Operation	Affected node	Type	Index	Parent node
INS	name_Private_System.String	MemberField	0	Seminar

Table 1. The difference (ε_1) between Revision A and Common ancestor

The index of the edit operation that inserts method *setNumber* into RevisionB has to be increased, because under class node *Seminar* in the first position the declaration node of the *name* attribute is inserted that shifts the positions. The correct indices of the nodes are *name*: 0, *number*: 1, and *setNumber*: 2.

Operation	Affected node	Type	Index	Parent node
INS	setNumber_Public_System.Int32	MemberMethod	1	Seminar
INS	In_n_System.Int32	ParameterDeclExpr	0	setNumber_Public_System.Int32_Parameters
INS	Instructor	TypeDeclaration	1	Global_Types
INS	name_Public_System.String	MemberField	0	Instructor

Table 2. The difference (ε_2) between Revision B and Common ancestor

After the script conversion, in the merge phase the updated edit scripts are applied to both sides, which performs the change propagations. In this approach the edit scripts are used as inputs by the merging algorithms to control the merge. The merge can be considered as tree patching, it executes the modifications described by the edit operations. Formally, $S_2=S_1+\varepsilon_2'$ and $M_2=M_1+\varepsilon_1'$. The model in the database (M_2) and the AST in the memory (A_2) have to contain the same elements both syntactically and semantically. This synchronized state (S_2) is stored as a common ancestor for the next iteration.

It is also important to note that the synchronization works for round-trip engineering as well. For instance, if the model M_0 in Fig. 3 is not defined, and, consequently, the AST model (M_1) is empty, the difference algorithm detects that the whole model M_2 must be built from scratch during the synchronization, and this should also work in the opposite direction (i.e. code generation).

4.2 A Platform-Specific Solution

One of the platforms supported by VMTS is Windows Mobile that solution is presented in Fig. 6. VMTS uses the CodeDOM technology [25] as a language independent model representation of the source code, and this means that the code generation is just a syntax tree composition.

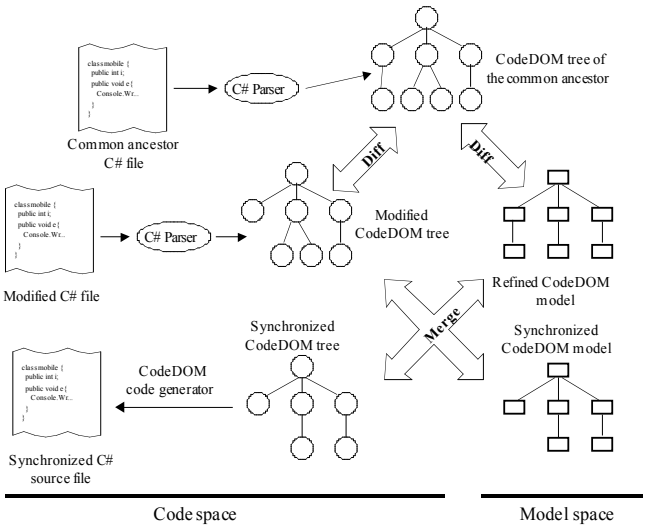

Fig. 6. Synchronization in the Windows Mobile platform

[26] introduces how VMTS generates source code using model transformation methods and CodeDOM metamodel. PIMs are transformed by model processors into CodeDOM instance models. The code generation technology of CodeDOM has several internal pretty-printers i.e. for C#, VB.NET, and C++ languages, but it can be extended to support other languages.

We have created a demonstrative proof-of-concept implementation of the proposed approach. The open source C# parser NRefactory [27] is used, that is capable of parsing, tokenizing, and building abstract syntax tree. The CodeDOM object tree in memory is built by an AST visitor class provided with NRefactory. The CodeDOM tree and the CodeDOM model (PSM) in the database are traversed and ordered, labeled trees are built from them, which are easy to process due to their homogeneous structure. The labels of the nodes are the types of the AST elements to help the matching.

Each node wraps an element from the parsed CodeDOM tree or from the model. The tree can be traversed in a general way, each special attribute of an AST class are explicated as node children.

The tree differencing and edit script computing are currently performed by the algorithm contributed in [14], because it was easy to implement and allows using heuristics (node similarity) during the matching, but we work on an algorithm that is more suitable for our requirements for AST matching. The edit scripts are transformed by our algorithm that works as mentioned previously. There are two different edit script execution units in the merger: one patches the CodeDOM object tree and the other handles the CodeDOM model in the database. These execution units are explicitly controlled by edit scripts.

Reliability criterias of the merging approach can be the following: (i) all changes must be detected, (ii) all must be propagated to the other side, (iii) syntactical correctness is ensured, (iv) there are no compile errors after the merge i.e. semantical correctness, (v) model consistency remains (vi) program (business) logic is unharmed, (vii) proper conflict handling e.g. automatic decision making, overlapping changes, identical changes (duplication) detection and solution, reorder the newly inserted nodes. Criteria iv, vi and some of vii have not been addressed yet. General limitations of syntactical-based techniques are that they are unable to detect undeclared variables, since the code is still syntactically correct. Some object-oriented behaviour i.e. the dynamic binding cannot be resolved by only syntactical approaches. Certain techniques and related problems are enumerated in [17].

We tried the synchronization on a source code, which was auto-generated by VMTS plugin [28] generator. The source file contained 2100 lines of code, 55 classes, its size was 104 kb, and the number of the tree elements affected by the differencing was above 7000. We made two copies of that file to create the common ancestor and the two revisions. Some changes were made to the revisions without making a conflict: a method body was removed and an extra class, with a method and several statements in it, was added.

Currently one file is generated directly from a PSM in VMTS. We also tested the synchronization of the AST model with the generated file. To simulate the changes of the PIM, for example a class diagram, the changes in the AST model and in the file affected the attributes, the methods, and method parameters. After the synchronization, the changes were propagated bi-directionally. During these simple studies, all criteria have been met. The expected results were that the edit scripts exactly reflect the incorporated changes, and all of them were propagated. Further studies require more complex examples with the PIM-PSM model synchronization, which is currently a subject of our research.

4.3 Limitations and Shortcomings

The drawback of reading source code into AST to manipulate is that the subsequent serializing looses the formatting and comments unless the parser, the pretty-printer and the AST take care of formatting information and comments. Although CodeDOM contains comment node, only namespace, class and class member nodes handle the comments. Fortunately the chosen parser reads the comments but stores them separately from the AST nodes. However, a complete synchronization approach should deal with the changes in the comments as well. This depends on the parsing technology and the design of the AST.

CodeDOM has some other limitations; it does not support all of the language elements introduced by C#. Using code snippet nodes the need of the unsupported elements can be bypassed. It works well on files, which are generated from domain-specific models by VMTS. These limitations hardly reduce the applicability of our solution, because the model-driven development mainly relies on the models and not on the generated code. The changes are typically performed in the model, but the developer has the opportunity to carefully modify the code, as well.

Distinction can be made between the conflicts: there are low-level syntactical (e.g. order of AST nodes) or simple semantic (e.g. conflicts derived from variable renaming) conflicts and high level semantic conflicts. To resolve high-level semantic conflicts, knowledge of a developer is essential, for example, when the same logic is incorporated into both artifacts and deciding which one should be omitted. The reliable automation of the presented approach cannot be fully achieved until the higher level intentions behind them are not extracted. The correct conflict resolution requires human validation or in case of automation, the result should be validated before use.

5 Conclusions and Future Work

The presented synchronization involves structural model-code differencing and three-way tree merging. The main advantage is that in contrast to typical code generation approaches, it permits modifying the generated code and instead of losing the changes, they will be synchronized back to the models. The incremental approach is more effective and keeps the previous modifications within both sides. The modular design allows the model-model and in addition the code-code synchronization.

CodeDOM models are applied as PSMs, which enables easy code generation. The illustrated method with some minor changes can be a solution to the synchronization challenge of other platforms as well. We separated the specific algorithms from the general ones, the differencing and the edit script transformation algorithms that contain the core logic of the synchronization are reusable. For instance, to support Java platform, a parser to Java language has to be written, which is capable of building CodeDOM tree in memory or a converter is required that creates CodeDOM tree from the AST of a Java source. Code generation for Java can be performed by the J# code provider [29] of CodeDOM. Obviously any other language dependent or independent code syntax scheme can be used as PSM instead of CodeDOM.

VMTS facilitates effective modeling with easy domain-specific modeling language definition and the dependent customizable domain-specific visualization. The introduced model-code synchronization method affects not only the higher level parts of the classes i.e. methods and attributes, but also the whole source code space including statements. In the code-to-model direction, this approach is useless without model transformations, because the

complex CodeDOM model has to be transformed into domain-specific models in order to make the approach useable in the practice. One drawback is that if the granularity of the transformation rules is not fine enough, some small changes in the code will not affect the PIMs directly. The prevention of information loss during the model transformations bridging different levels of abstraction should be facilitated by tracing the transformations.

In contrast to text-based approaches, syntactical correctness is ensured, since the change propagation works on the ASTs of the selected language and the elements of an AST are based on the grammar of the language. Semantically correct synchronization and supporting more software platforms are the subject of our future work.

Acknowledgements

The fund of "Mobile Innovation Centre" has supported in part, the activities described in this paper. This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

The C# source code parsing into CodeDOM is performed by the free open source NRefactory parser [27].

References

[1] OMG, MDA Guide Version 1.0.1, document number: omg/2003-06-01, http://www.omg.org/docs/omg/03-06-01.pdf
[2] OMG UML 2.0 Spec., http://www.omg.org/uml/
[3] J. Sprinkle, "Model-Integrated Computing", IEEE Potentials, 23(1), 2004, pp. 28-30.
[4] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf, "A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering", Ninth Working Conference on Reverse Engineering, Washington, 2002, pp. 22-34.
[5] C. Enrique Ortiz, "Introduction to Symbian OS for Palm OS developers", http://www.metrowerks.com/pdf/IntroSymbianOSforPalmDevelopers.pdf
[6] Sun, Java 2 Platform, Micro Edition (J2ME), http://java.sun.com/j2me/index.jsp
[7] Microsoft Windows Mobile, http://www.microsoft.com/windowsmobile
[8] G. Rozenberg (ed.), Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol.1 World Scientific, Singapore, 1997.
[9] VMTS web site, http://vmts.aut.bme.hu
[10] P. Newcomb, "Abstract Syntax Tree Metamodel Standard ASTM Tutorial", OMG's Second Annual Architecture-Driven Modernization Workshop, Alexandria, USA, 2005.
[11] E. W. Myers, "An O(ND) difference algorithm and its variations", Algorithmica, 1(2), 1986 pp. 251-266.
[12] W. F. Tichy, "The string-to-string correction problem with block moves". ACM Transactions on Computer Systems, 2(4), November 1984, pp. 309-321.
[13] UNIX diff manual, web site: http://unixhelp.ed.ac.uk/CGI/man-cgi?diff
[14] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information", In Proceedings of the ACM SIGMOD International Conference on Management of Data, 25(2), Montreal, Quebec, June 1996, pp. 493-504.
[15] S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data", Proceedings of the 1997 ACM SIGMOD international conference on Management of data, 26(2), May 1997, pp. 26-37.

[16] S. Khanna, K. Kunal, and B. C. Pierce.” A Formal Investigation of Diff3”, Manuscript, University of Pennsylvania, 2006.
[17] T. Mens, “A State-of-the-Art Survey on Software Merging”, IEEE Transactions on Software Engineering, 28(5), May 2002, pp. 449-462.
[18] D.C. Oppen, “Prettyprinting” ACM Transactions on Programming Languages and Systems, 2(4), 1980, pp. 465-483.
[19] U. Asklund, “Identifying Conflicts During Structural Merge”, Proceeding of the Nordic Workshop on Programming Environment Research '94, Lund University, 1994, pp. 231-242.
[20] T. Lindholm, “A three-way merge for XML documents”, Proceedings of the 2004 ACM symposium on Document engineering, October 28-30, 2004, Milwaukee, Wisconsin, USA, pp. 1-10
[21] R. la Fontaine, “Merging XML files: a new approach providing intelligent merge of XML data sets”, In Proceedings of XML Europe 2002, Barcelona, Spain, 2002.
[22] W. Yang, “How to merge program texts”, Journal of Systems and Software, Vol. 27, No. 2, 1994, pp. 129-135.
[23] N. Niu, S. Easterbrook, and M. Sabetzadeh “A Category-Theoretic Approach to Syntactic Software Merging”. 21st International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, Sept. 2005. pp 197-206.
[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns”, Addison-Wesley, Massachusetts, 1994.
[25] Microsoft’s CodeDOM web site, http:// msdn2.microsoft.com/en-us/library/system.codedom.aspx
[26] L. Lengyel, T. Levendovszky, G. Mezei, B. Forstner, H. Charaf, “Metamodel-Based Model Transformation with Aspect-Oriented Constraints”, Electronic Notes in Theoretical Computer Science, Vol. 152, pp. 111-123.
[27] #develop web site: http://sharpdevelop.net/
[28] G. Mezei, T. Levendovszky, H. Charaf, “A Presentation Framework for Metamodeling Environments”, Workshop in Software Model Engineering, Montego Bay, Jamaica, 2005.
[29] Visual J# code provider web site: http://msdn2.microsoft.com/en-us/library/w5e3ax1a(VS.80).aspx