SOFTWARE ENGINEERING ENVIRONMENT FOR BUSINESS INFORMATION SYSTEMS

Alar Raabe
Profit Software AS, Türi 9, 11314 Tallinn, Estonia
Email: alar.raabe@profitsoftware.ee

Keywords: Model driven synthesis, meta-model extensions, software engineering, software engineering environment

Abstract: There is a growing need to reduce the cycle of business information systems development and make it independent of underlying technologies. Model driven synthesis of software offers solutions to these problems. This article describes a set of tools and methods applicable for synthesizing business software from technology independent models. This method and these tools are distinguished by the use of extended meta-models which embody knowledge of the problem domain and target software architecture of the synthesized software system by the use of the model conversion rules described using the combined meta-model and by the use of reference models of problem domains and sub-domains, which are combined and extended during the construction of software system descriptions. The difference of our method from other domain specific methods is the separate step of solution domain analysis and the use of meta-model extensions. This study has been done in the context of developing product-line architecture for insurance applications.

1 INTRODUCTION

Today's business processes have become more dependent on the software, and at the same time are changing very rapidly in response to the market changes. It is characteristic of business information systems that initial results from software development should be delivered with a very short delay, and when the business volume grows or the business processes change, the system must be able to grow along, without impeding the business process (e.g. without major reimplementation effort). Usually to achieve different qualities of service (e.g. scalability, reliability, security, etc.) required for business information systems different implementation technologies have to be used, or several different implementation technologies have to be combined.

At the same time implementation technologies of software systems are also developing at fast pace, often without offering backward compatibility. To avoid becoming tied to a legacy software, which requires expensive measures to maintain and to take advantage of the most recent developments in the implementation technologies and base software (e.g. application servers, operating systems, etc.), business information systems should be reimplementable quickly, using a different implementation technology.

In addition, because of this fast change in the implementation technologies and the need for change of underlying implementation during the life cycle of business information system, the main body of reusable software assets of an enterprise should be independent of specific implementation technologies.

These problems are addressed by the model-based approaches to the software development (e.g. model-based software synthesis (Abbott et al., 1993), model-based development (Mellor, 1995), model driven architecture (MDA) (OMG, 2001a), etc.), where the main artifact of software development is implementation technology independent model of a required software system, which becomes the source of concrete implementation created through synthesis or generation.

When convergent engineering principles of software design (Taylor, 1995) are applied, then analysis and design will produce artifacts that are easily mapped into the implementation constructs, making the automatic generation of implementation easier. This will be possible if the analysis and

design models are based on a meta-model, which is rich enough to capture details needed to synthesize implementation (Melnikov, 1990).

We treat the development of the business information systems similar to the domain oriented application development technologies (SEI and Honeywell), where business in general is treated as a large general domain containing several more specific domains (business areas), which refer to the common elements from the general business domain.

This article describes a set of tools and methods applicable for synthesizing business information software from technology independent models. These tools are distinguished by the usage of extended meta-models (Raabe, 2002), which embody knowledge of problem domain and target software architecture, by the usage of model conversion rules described using the combined meta-model, and by the usage of reference models of problem domains, which are extended during the construction of descriptions of the software system.

The problems analyzed in this article are:

- creation and usage of reference models during the development process;
- composition of reference models;
- steps of software process for model-based software development;
- parts of software engineering environment (SEE) targeted to usage of models.

This article covers the studies done in the context of developing a product-line architecture (Parnas, 1976 and Bass, Clements & Kazmann, 1998) for a family of insurance applications that applies principles of convergent engineering (Taylor, 1995) and the model driven approach (Abbott et al., 1993 and Mellor, 1995) to the insurance software production, developed under the guidance of the author.

Because insurance as an example of the problem domain is sufficiently complex, we assume that the techniques working in this domain would be applicable to other domains as well.

2 USAGE OF MODELS IN SOFTWARE ENGINEERING

Next we review the usage of models in the traditional methods of producing business information systems.

2.1 Definitions

We will use the following definitions from UML:

- *domain* is an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area (OMG, 2001b);
- *model* is a more or less complete abstraction of a system from a particular viewpoint (Rumbaugh, Jacobson & Booch, 1999), or *model* is an abstraction of a physical system with a certain purpose (OMG, 2001b).

We assume that domains may themselves contain more specific sub-domains, i.e. between domains can exist a generalization relationship (Simos et al., 1996). Based on this generalization relationship, domains form a taxonomic hierarchy.

We extend the meaning of the model to represent not only abstractions of physical systems, but also abstractions of logical systems.

Additionally, we introduce the following definitions:

- *domain model* is a body of knowledge in a given domain represented in a given modeling language (e.g. UML);
- *problem domain* of a software system is a domain which is the context for functional requirements of that software system;
- *solution domain* of a software system is a domain which describes the implementation technology of that software system;
- *reference model* is a representation of knowledge about the problem domain combined with the standard solutions for standard problems in that domain;
- *analysis model* is a model of a software system which contains elements from the relevant problem domain models and is essentially a combination and specialization of relevant problem domains for specific needs of a given software system specified by the set of functional requirements for the system;
- *implementation model* is a model of specific implementation of some software system which contains elements from the relevant solution domain models and is essentially a combination and specialization of relevant solution domains for specific needs of a given software system specified by the set of non-functional requirements for the system;
- *combination of models* is an operation which makes the elements of combined models available to the resultant model (e.g. in UML composition of models, importing of models, and inheritance of models (OMG, 2001 and Rumbaugh, Jacobson & Booch, 1999)).

Both the problem domain and the solution domain of a software system may contain several more specific sub-domains (Simos et al., 1996).

Problem domain is constructed according to the set of functional requirements to the software system, and solution domain is constructed according to the set of non-functional requirements to the software system.

We use the term *implementation model* instead of the design model to stress that this model represents not only the logical level of design, but the design of the software system for the specific combination of solution domains – a specific implementation.

We are interested that reusing the reference models during the analysis and design phase of software development will result in reusing the implementation of the same reference model during the implementation phase.

Similarly, we are interested in reusing the analysis model of a software system when non-functional requirements, and accordingly the implementation model, change.

2.2 Usage of models in the traditional software development process

A traditional software development process uses a generic meta-model, and both the system and its implementation are modeled using the same meta-model. The relationships between the generic meta-model and the models created in the traditional software development process are shown in Fig. 1.

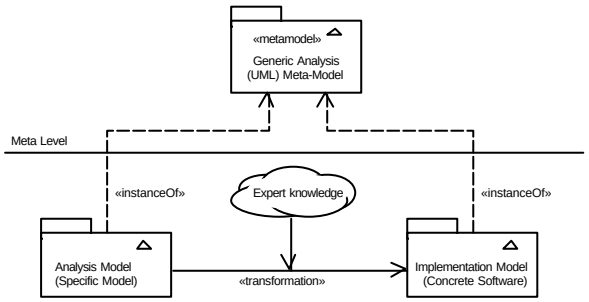

Figure 1: Traditional usage of meta-models and models

The problem with this approach is that the analysis model contains implicitly parts of the domain models of all the domains, which the given software system concerns (e.g. analysis model for insurance policy management and claim handling system for life insurance agency contains parts from the life insurance domain, parts from the claim handling domain and might contain also parts from the accounting and money management domains). Similarly, the implementation model of a specific software system contains parts from architecture models inherent to the chosen implementation technology.

Because problem domain model elements in the traditional analysis model, which are not specific to the given system, are intermixed with the model elements specific to the given system (e.g. describe the specific business situation and business processes), domain modeling effort made in the context of one software system is very hard to reuse for other software systems in the same domain, or when the given system must be changed because of the changes in the business domain.

Because the transformation from the analysis model to the implementation model is an informal one-way transformation, which produces the implementation model where the analysis model elements are intermixed with the elements of the solution domain, the results of the analysis effort are difficult to reuse when the given system must be changed or reimplemented, using different implementation technologies because of the changes in the non-functional requirements.

2.3 Software development process with extended meta-models

As described in (Raabe, 2002), one way to make the analysis and design processes more effective and guide the analysis in a specific domain, we propose using the extended analysis meta-models, which embody the domain knowledge and reference models, which are the results of partial analysis of a given problem domain, as a starting point for the analysis and design processes.

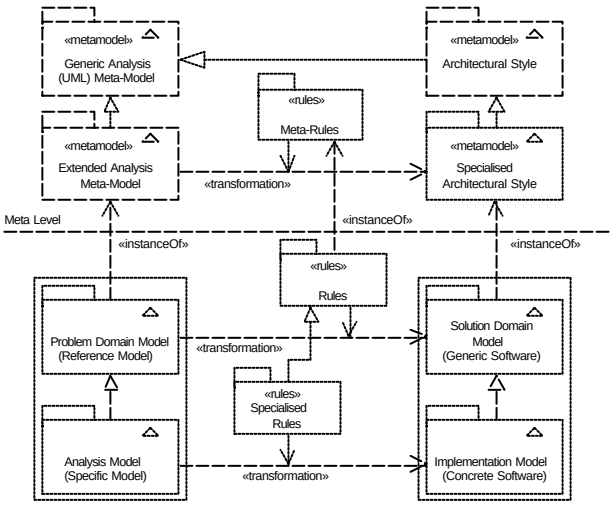

Figure 2: Proposed usage of extended meta-models and reference models

This enables us to define precise transformation

rules between several levels of models (Peltier, Ziserman & Bézivin, 2000 and Lemesle, 1998) usable in the software development process for synthesizing the implementation model from the analysis model.

The relationships between the different models and the transformation rules are shown in Fig. 2.

When we have separated the analysis model of traditional methods into the analysis model of a given software system and a set of problem domain models, and similarly, the implementation model into the implementation model of a given software system and a set of solution domain models, it will be possible to reuse the analysis efforts when reimplementation of a given software system is required.

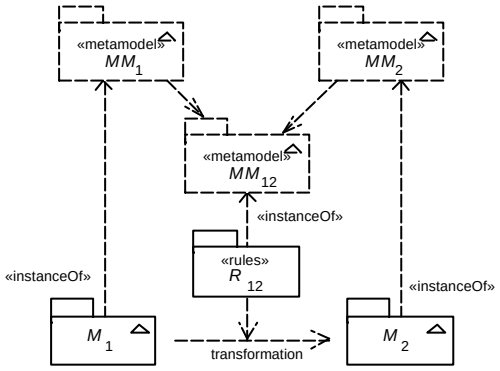

Figure 3: Need for combined meta-models for model transformations

As shown in Fig. 3, to describe transformations between different models, which possibly use different meta-model extensions, we need to combine source and target meta-models and their extensions to represent the transformation rules, which need to access concepts from both meta-models.

A set of operations for combining of meta-model extensions is described in (Raabe, 2002). There we propose to extend semantics of UML with the model combination operations and describe techniques to achieve interoperability of meta-model extensions, and to allow isolation of the developed model from the changes in the meta-model.

3 REFERENCE MODELS

A reference model is a representation of knowledge about the problem domain, which are results of partial analysis of a given domain combined with the standard solutions for standard problems in that domain.

To make the analysis process more effective and guide the analysis in a specific domain, we have to provide reference models as a starting point for the analysis and design processes, by giving a set of ready-made design decisions applicable to the given problem domain.

As a result of nested problem domains there will be a need to create specialization hierarchies of reference models.

3.1 Creation of reference models

There are some considerations that have to be taken into account when reference models are created. These considerations are meant to facilitate reusing and combining the reference models.

The first one concerns the usage of classes of roles instead of classes of objects. A role is a context-specific view of an object (see role object design pattern in (Bäumer, 2000)). A role class is an element of a model which represents the classifier in some other model. Role classes are similar to the connection mechanism called “roles” used in UML to connect classifiers to the collaborations.

The second consideration is to clearly mark the extension or variability points in the model.

The third one concerns the isolation of the possible variable functionality or behavior by reification, for example, using factory and strategy patterns.

The last one is the clear clustering of model elements. When selectively reusing model elements from the reference models it is not possible to select arbitrary model elements, but reuse has to happen by the clusters of model elements.

3.2 Role-based modeling

To describe the difference between role clasess and object classes, let us see the example (Fig. 4), where we model the classification of persons to customers and beneficiaries.

When using the static disjoint classification, objects cannot change their classes during their lifetime and each object can only belong to one class. If we try to represent situation where same person can be a customer and a beneficiary at the same time, we will need an additional identity mechanism, which has to connect instances of these classes together.

When using the role classes, there is a possibility that several roles will represent the same object at the same time. This set of roles can change during an object’s lifetime.

Role classes represent the multiple dynamic classification (Rumbaugh, Jacobson & Booch, 1999)

of objects, where objects may acquire and loose classes during run-time.

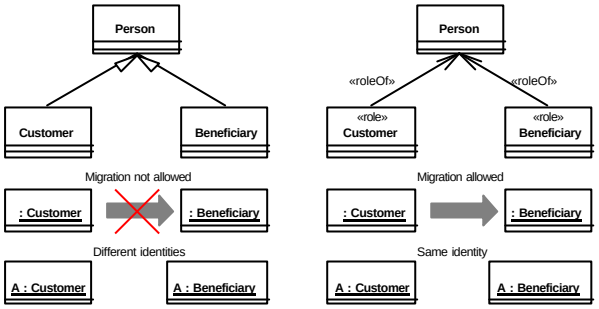

Figure 4: Difference of role classes from object classes

To facilitate cascading combinations of models, role classes may represent other role classes, as shown in Fig. 5.

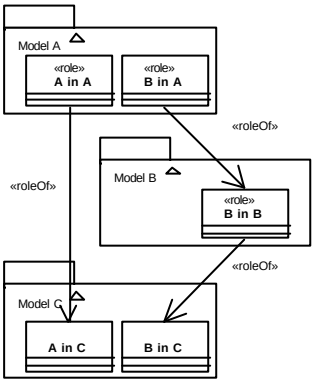

Figure 5: Usage of role classes to support reuse

When role-based modeling is used for reference models, it is possible to (re)use the reference model by associating the model classifiers with the role classes of reference models.

3.3 Combination of reference models

Assuming that the used models are represented as UML models, we can use *containment*, *importing*, or *multiple inheritance* of models, to achieve model combination (OMG, 2001) as shown in Fig. 6.

When using model containment to combine different reference models, combined model elements are encapsulated and not visible to the combining model or each other. To be able to "see" those elements in the combining model, they have to be either imported or the combining model should be a specialization of all the combined models. Therefore containment is not in practice suitable for combining the reference models.

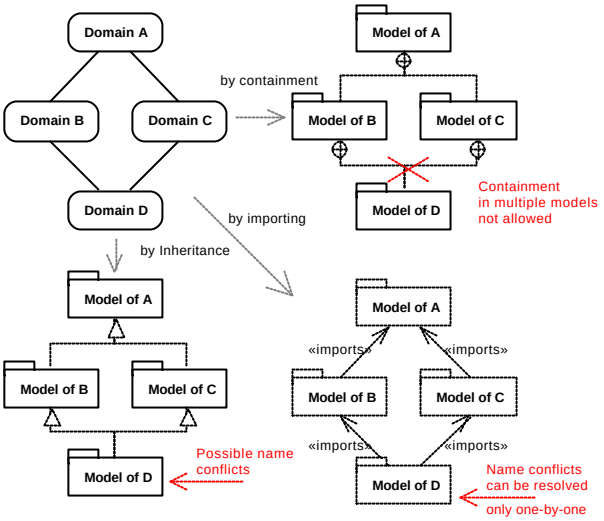

Figure 6: Model combination techniques in UML

When using model importing for combining different reference models, the dependency with the stereotype «imports» in the UML describes access permission, i.e. that an importing model imports all the elements with sufficient visibility from the supplier models, including elements of models imported by the supplier models that are given public visibility in the supplier. Because it is not possible to build model hierarchies with importing, it has only limited value as a mechanism for combining the reference models.

When using multiple inheritance of models to combine different reference models, it is possible to construct taxonomic hierarchies of models, because a model can have generalizations to other models. The mechanism of constructing the description of a specific model out of more general models is inheritance, i.e., the public and protected elements owned or imported by more general models are also available to its children – more specific models, and they can be used similarly to any element owned or imported by the child models themselves.

Elements inherited from other models due to generalization retain their name and extend the namespace of the inheriting model. By default, inherited elements have the same visibility both in the child model and in the parent models. It is not possible to change the name or visibility of inherited elements in the inheriting model.

Problems of combining models in UML are as follows:

- name conflicts between elements from different models;
- conflicting model elements (conflicting features, relationships and constraints);

- cluttered resultant model (because all of the combination methods in UML are only additive);
- difficulty in changing the used meta-model extensions of the model.

4 SOFTWARE PROCESS STEPS

We propose that software development process for problem-oriented software will use both meta-model extensions and reference models and contains the following steps:

1. *Problem domain analysis*, which produces meta-model extension(s) specific to a given problem domain and a set of reference models, defined in terms of these meta-model extensions. The results of the problem domain analysis are reusable for all the systems which share the same problem domain.
2. *Solution domain analysis*, which produces meta-model extension(s) specific to a given solution domain and an architecture model (architectural style) defined in terms of these meta-model extensions. The results of the solution domain analysis are reusable for all the systems which share the same solution domain.
3. *Generic solution design*, which produces models of generic solutions for a given solution domain. The results of the generic solution design are reusable for all the systems which share the same solution domain.
4. *Implementation of architecture*, which produces specific software artifacts needed to implement the designed generic solutions in a given solution domain.
5. *Problem to solution mapping design*, which produces transformation rules for problem domain models to solution domain models transformation on different model levels. The results of problem to solution mapping design are reusable for all the systems which share both the problem and solution domain.
6. *Specific problem analysis*, which uses products of problem domain analysis (meta-model extensions and reference models) and produces the model of a given software system.
7. *Synthesis of a specific system*, which uses transformation rules developed during problem to solution mapping design, the chosen implementation of architecture and produces specific implementation of a system.

The software process with the process steps relationships to the used and produced models is shown in Fig. 7.

Steps 1-5 are independent of the specific software system and the investments needed to perform these steps can be spread over the product line or a family of systems (Parnas, 1976 and Bass, Clements & Kazman, 1998), which share the same problem domain and solution domain.

The difference between our approach and domain modeling approaches presented in (SEI and Honeywell) is that we propose to separate the problem domain analysis and solution domain analysis, and require that the analysis result of both domains will be presented as meta-model extensions. We also prescribe a separate design step, where transformation rules from the problem domain to the solution domain on several model levels will be produced.

These reference models must be specialized to create concrete analysis and design models for a given problem. As a result of nested problem domains and need for interoperability between the specific reference models there will be a need to create specialization hierarchies of reference models. To be able to further specialize the model, certain aspects exist during the model construction that have to be taken into account: modeling should be based on the roles (or aspects) of business objects, and variable parts of the model should be isolated.

Catalysis methodology (D'Souza & Wills, 1999) describes a specialization of models which is purely additive. Our experience shows that when trying to compose specific reference models to produce a model of a required system, the additive model specialization is not enough. It tends to produce models that contain unnecessary elements. To avoid this, we propose to use techniques applicable during the specialization of the model, like overriding the model elements and removing unused elements (Raabe, 2002).

5 PARTS OF SOFTWARE ENGINEERING ENVIRONMENT

Software engineering environment that supports the described technology of software engineering consists of the following parts:

- repository of models which implement meta-model extensions and model combination operations;
- tools for manipulating the models and extended meta-models;

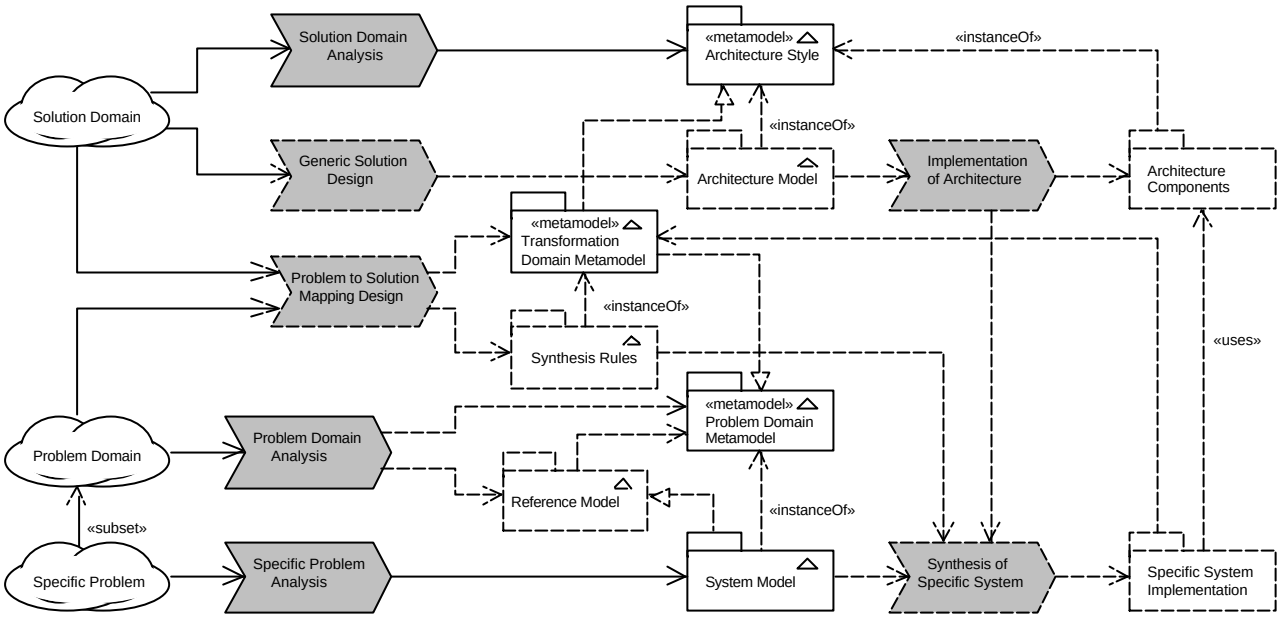

Figure 7: Model-oriented software development.

- reference models of the needed problem domains;
- changeable implementations of base architectures that correspond to different implementation technologies and embody generic implementations of reference models stored into the repository;
- rule-driven generators which implement the model transformations.

6 PRACTICAL APPLICATION

A practical application of the presented techniques and tools for software engineering was developed under the guidance of the author at the Progressive Financial Technologies Ltd. during 1995-2000 to develop insurance software sold under the registered trademark Once&Done®.

Once&Done® software forms a product-line architecture (Parnas, 1976 and Bass, Clements & Kazman, 1998) of insurance applications based on the convergent engineering principles (Taylor, 1995). All members of Once&Done® product-line are based on a set of reference models of insurance business, and on the common object-oriented architecture.

Software environment for producing Once&Done® product-line members consists of the following:

- *Models* consisting of the insurance specific extension of the meta-model of the traditional object-oriented analysis and reference models of the insurance domain organized according to the main elements of the insurance domain (like party, policy, insurable, coverage, property and casualty insurance, life insurance, etc.).
- *Object-oriented framework* consisting of elements which implement technical (base) services for building object-oriented business software – an environment for business objects and generic implementation of insurance domain models and the related insurance functionality.
- *Process* containing the description of steps and tasks required to create a member of the product-line, and based on the object-oriented paradigm. The goal is to support the creation of the insurance software based on the Once&Done® product-line architecture, maintaining the quality and predictability, identification of reusable elements and the accountability (visibility) of the process.
- *Tools* containing facilities for using the framework and models according to the process to produce members of the product-line. A central tool is the Once&Done® Specification Environment (OD-SE), which implements the extended analysis and design meta-model. Additionally, various generators

permit us to generate concrete implementations of business objects based on the information in the OD-SE repository. OD-SE allows us to connect several OD-SE repositories, enabling one to create members of the product-line by combining multiple existing models.

Because the whole development cycle of software is based on the same model (according to convergent engineering principles (Taylor, 1995)):

- software engineering process is simplified and the total amount of work is reduced;
- gaps between business processes and their supporting software are minimized;
- modifications to the business processes and the supporting software are easily coordinated.

An analyst and a designer create a model of a software system, using the provided reference models. As compared to the traditional universal modeling methods, where an analysis usually starts at a blank page, this makes the analysis and design processes easier and shorter. When using the model inheritance to combine the needed reference models into the model of the required system, we assure that all the changes of reference models are inherited to the model of the software system, making it easy to keep all the systems based on the same reference models consistent.

Insurance products that can be composed of elementary parts to cover certain risks involve complex business rules and compose a large domain, which must be separately modeled before the systems supporting these products can be built. A combination of meta-model extensions suitable to describe insurance business processes and insurance products in Once&Done® allows a description of business processes and insurance products as an integral part of insurance systems model. When compared to the traditional universal modeling methods, this reduces the number of models that must be constructed, makes models smaller, and makes the transformation from the analysis models to design models easier.

Changing the meta-model extensions that describe implementation architecture allows the generation of different implementations of the insurance system out of the same model. This has been tested by changing the implementation architecture of the same insurance system from client-server architecture with a fat client to a three-tier server centric architecture with a thin client, without changes to the insurance system model.

7 RELATED WORK

Domain engineering in (SEI) separates the software engineering process into two larger parts: domain engineering and application engineering.

The difference of our method is that we clearly separate the step of architecture engineering, which consists of solution domain analysis and implementation of architecture, from the domain engineering, which should be performed before the application engineering. The results of this step are the corresponding extensions of meta-model, which allow to describe transformation rules for transforming the application model into implementation with the given architecture style, and the architecture framework.

Similar problems of model combination are dealt with in the Generic Modeling Environment (GME 2000) (Ledeczi, Volgyesi & Karsai, 2001 and Ledeczi et al., 2001). GME 2000 uses three additional model operators to support the model composition: equivalence of classes, interface inheritance and implementation inheritance of models. The equivalence operator constructs a union of two different classes. The proposed new model inheritance operators define the fixed selection criteria for model elements, which are taken from the source model by one of these operators. Interface inheritance takes all the associations and compositions, where the source model element is in the role “contained”. Implementation inheritance takes all the attributes and compositions, where source model element is in the role of “container”.

Our experience shows that in real world applications it is necessary to allow more complex selection criteria for model elements, which are inherited from the base models.

Lately the MDA initiative from OMG (OMG, 2001a) has been establishing modeling standards needed to develop supporting tools for platform independent application description.

Techniques and tools presented in the article are in line with MDA and useful when the MDA approach is applied to the development of large-scale business systems.

8 CONCLUSIONS

We have shown that for supporting the model driven synthesis of software, there exists a need for combining models and meta-models. In the case of model inheritance used for combining the models, operations of overriding, replacing and deferring of inherited elements are needed.

We have also shown that usual analysis and

design techniques do not produce the reference models that are suitable for the model combination. We have proposed a modeling technique that uses role-based modeling, clear identification of variation points (through the usage of feature analysis), separation of functionality and explicit clustering of model elements, to produce reference models which are easily used during the modeling of specific software systems.

Finally, we have introduced new steps in the software process – solution domain analysis and problem to solution mapping design. In addition, we require that the results of the problem and solution domain analysis be presented as meta-model extensions.

9 ACKNOWLEDGEMENTS

Author wishes to acknowledge gratefully Profit Software Ltd. (Finland) and the Estonian Science Foundation for their support (Grant 4721).

Author wishes to thank late Boris Tamm for discussions on the subject and many useful suggestions for improving this paper.

10 REFERENCES

Abbott, B., Bapty, T., Biegl, C., Karsai, G., Sztipanovits, J., 1993. Model-Based Software Synthesis, *IEEE Software*, May, 10 (3), 1993, pp.42-52.

Mellor, S. J., 1995. Reuse through automation: Model-Based Development, *Object Magazine*, September 1995.

OMG, 2001a, *Model Driven Architecture*, OMG 01-07-01, ftp.omg.org/pub/docs/ormsc

Taylor, D. A., 1995, *Business Engineering with Object Technology*, John Wiley & Sons, New York.

Melnikov, I., Raabe, A., 1990, Expert System Based Computer Aided Software Engineering (CASE) Environment. In *Proceedings of 5. Symposium "Grundlagen und Anwendungen der Informatik" 6.-8. Februar 1990*, Wissenschaftliche Tagungen der Technischen Universität Karl-Marx-Stadt 6/1990, pp.180-188.

SEI, Domain Engineering: A Model-Based Approach, www.sei.cmu.edu/domain-engineering

Honeywell, Domain-Specific Software Architectures, www.htc.Honeywell.com/projects/dssa

Raabe, A., 2002, Techniques of combination of metamodel extensions, *Proceedings of the Estonian Academy of Sciences, Engineering*, 8 (1), 2002, pp. 3-17.

Parnas, D., 1976, On the Design and Development of Program Families. *IEEE TSE*, 2 (1), 1976, pp.1-9.

Bass, L., Clements, P. and Kazman, R., 1998, *Software Architecture in Practice*, Addison-Wesley.

OMG, 2001b, *OMG Unified Modeling Language Specification Version 1.4*, OMG 01-09-67, ftp.omg.org/pub/docs/formal

Rumbaugh, J., Jacobson, I., and Booch, G., 1999, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, Massachusetts.

Simos, M., Creps, D., Klinger, C., Levine, L., and Allemang, D., 1996, *Organization Domain Modeling (ODM) Guidebook, Version 2.0, Technical Report for STARS*, STARS-VC-A025/001/00, June 14, 1996.

Peltier, M., Ziserman, F., Bézivin, J., 2000, On Levels of Model Transformation, In *XML Europe Conference, Paris France, June 2000*, www.gca.org/papers/xmleurope2000.

Lemesle, R., 1998, Transformation Rules Based on Meta-Modeling, In *Proc. of EDOC'98, 3-5 November 1998, La Jolla, California, USA*, 1998, pp. 113-122.

D'Souza, D. F., Wills, A. C., 1999, *Objects, Components, and Frameworks with UML, The Catalysis Approach*, Addison-Wesley, Reading, Massachusetts.

Bäumer, D., Riehle, D., Siberski, W. and Wulf, M., Role Object, In *Pattern Languages of Program Design 4*, Addison-Wesley, Reading, Massachusetts, 2000, pp. 15-32.

Kendall, E. A., Role Model Designs and Implementations with Aspect Oriented Programming, In *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, ACM Press, November 1999.

Ledeczi, A., Volgyesi, P., Karsai, G., 2001, Metamodel Composition in the Generic Modeling Environment, In *European Conference on Object-Oriented Programming (ECOOP'2001). Workshop on Adaptive Object-Models and Metamodeling Techniques, Budapest (Hungary), June 2001*, adaptiveobjectmodel.com/ECOOP2001/submissions

Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., and Maroti, M., 2001, On Metamodel Composition", In *IEEE CCA 2001, Mexico City, Mexico, September 5, 2001*, www.isis.vanderbilt.edu/publications/archive