Formalizing Model Consistency Based on the Abstract Syntax

Frank Trollmann, Marco Blumendorf, Veit Schwartze and Sahin Albayrak
DAI-Labor
Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
{frank.trollmann, marco.blumendorf, veit.schwartze, sahin.albayrak}@dai-labor.de

ABSTRACT
In this paper we define a notion to describe consistency within and between models, which has been identified as important issue when using model-based tools. We introduce the abstract syntax of models as attributed typed graphs and define a formalism of consistency based on this formal description. The application of the formalism is illustrated by an example.

Author Keywords
Model Driven Engineering, Abstract Syntax, Consistency

ACM Classification Keywords
D.2.4 Software/Program Verification: Formal methods

General Terms
Theory

INTRODUCTION

With the increasing utilization of models to address design as well as runtime issues and the quickly growing number of new domain specific modeling languages, model consistency becomes an important issue. Tools and runtime systems read, process, execute and change models and need to take care of consistency issues that can even span multiple related models noted in different languages. This raises an urgent need for a general understanding of model consistency and new ways to ensure this consistency throughout tool chains and within runtime systems.

In this paper, we introduce a formal understanding of consistency. This notion of consistency is based on the abstract syntax of models and the distinction between abstract and concrete syntax as detailed in the next two sections. Afterwards a definition of a way to describe this abstract syntax is presented and it is shown how this description can be used to define internal consistency as well as consistency between models. Thereafter, the formalism is illustrated in an example and the final section concludes the paper and hints to future work.

PROBLEM STATEMENT

With increasing popularity of model based and model driven approaches more and more tools for handling models emerged. In order to be handled correctly by these tools models need to be consistent. We distinguish two kinds of model consistency. Intra-model consistency is the consistency of a model in itself and means that a model is an element of the correct modeling language. Inter-model consistency is the consistency of several models with respect to each other. This kind of consistency requires the information contained in these models to not contradict each other.

One example of an environment that depends on model consistency is a runtime modeling framework. In such a framework a software application results from the interpretation of a set of models. In order to interpret them the framework has to rely on the conformance of each model to its modeling language. If a model is not consistent with respect to its modeling language (intra-model consistency) it cannot be interpreted. An inconsistency between several models (inter-model consistency) can lead to unexpected behavior.

However, due to the variety of models and the heterogeneity of domain-specific modeling languages the definition of consistency is a challenging task. In order to express consistency between models of different modeling languages a common formalism is required. This formalism needs to be able to express intra- and inter-model consistency and has to cope with the heterogeneity of models.

We address these problems by the introduction of the abstract syntax of a modeling language. If the abstract syntax of several models is expressed in the same formalism it can be used to define inter-model consistency between these models. The following section introduces the notion of abstract and concrete syntax of a modeling language and interrelates both. Afterwards a common notion for an abstract syntax and a way of defining model consistency is described.

ABSTRACT AND CONCRETE SYNTAX

In programming languages there is a distinction between abstract and concrete syntax. The concrete syntax of a programming language is the version the programmer works on. The abstract syntax is a representation of the program that can be automatically processed. During the compile process the concrete syntax is parsed into the abstract syntax. Optimization and analysis is done on the abstract syntax.

This concept can be borrowed by modeling languages. The concrete syntax of a model is the version that is handled by its designer. In order to be processed by a modeling tool the concrete syntax is parsed into the abstract syntax. The abstract and concrete syntax can be regarded as two models that represent the same aspects of a common system under study.

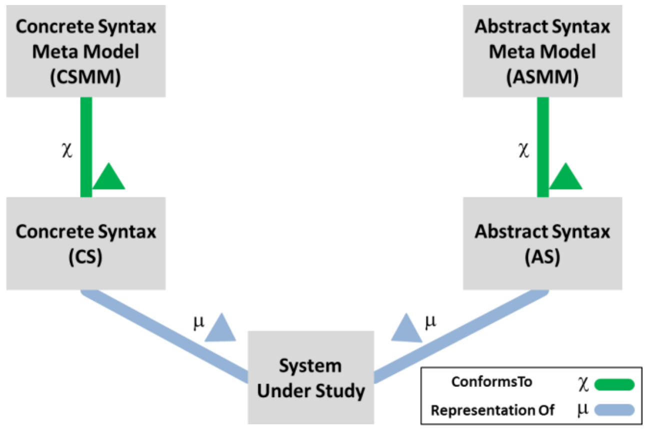

Figure 1. Concrete and abstract syntax and the transformation between them.

This situation is depicted in Figure 1. In this figure we use the terminology introduced by Favre [1]. According to this terminology a metamodel describes a modeling language which consists of a set of models. In Figure 1 both the abstract and concrete syntax are models representing the same system under study but conforming to different metamodels.

If the abstract syntax of two models is expressed in the same formalism it is easier to describe consistency between them. The following section introduces a common basis for the definition of the abstract syntax and a definition of intra- and inter-model consistency on this basis.

DESCRIBING ABSTRACT SYNTAX AS ATTRIBUTED TYPED GRAPHS

In programming languages the abstract syntax of a program is often represented by an abstract syntax tree, containing the functions and expressions of the program.

In fact, most models can be represented by a graph-based structure. For instance many UML tools offer an option to export the models in an XMI format, which can be seen as a tree of XML nodes and thus a graph-like structure. Hermann et. al. use a form of attributed typed graphs to represent the abstract syntax of class and sequence diagrams [2]. Limbourg et. al. use directed attributed typed and labeled graphs as a general representation of models and foundation for their transformational development [3].

A graph consists of nodes and edges between these nodes. In attributed graphs the nodes and edges can be annotated with attributes. An attributed graph can be typed with respect to a type graph. The type graph is an attributed graph that determines possible node and edge types. Typing is represented by a mapping between the typed graph and its type graph. An attributed graph with a morphism to a type graph is called an attributed typed graph (AT-Graph). A formal definition of AT-Graphs is given in [4].

We choose AT-Graphs as the formal basis of the abstract syntax. The language of this abstract syntax consists of a set of AT-Graphs. Its metamodel, the abstract syntax metamodel (ASMM), describes this set. We define that for a given modeling language all models are typed over the same type graph. This means the abstract syntaxes of models for one modeling language use the same set of node and edge types.

Accordingly, the ASMM of a modeling language contains the common type graph. In some cases not all graphs typed over this type graph are considered valid models. Most modeling languages have additional structural requirements. For instance a Finite State Machine needs an initial state to be valid. In order to express these requirements the ASMM contains an additional set of graph constraints as defined by Ehrig et. al. [5]. An atomic graph constraint consists of a graph morphism $a : P \rightarrow C$. A graph G fulfills this constraint if every occurrence of P inside of this graph also contains the structures defined in C. A graph constraint is a boolean formula over atomic constraints. A formal version of the ASMM is given in Definition 1.

Definition 1. An abstract syntax metamodel $asmm = (TG, cons)$ consists of an attributed type graph TG and a set of attributed typed graph constraints $cons$ that are typed over TG.

The ASMM describes the abstract syntax modeling language as the set of all attributed typed graphs that are typed over the type graph *TG* and fulfill the graph constraints *cons*.

Intra-model consistency is defined as the conformance of a model to this metamodel. If the model is correctly typed and fulfills the constraints it is conform to the metamodel and is called intra-model consistent. This is formally defined in Definition 2.

Definition 2. A model m in abstract syntax is intra-model consistent with respect to its metamodel $asmm = (TG,cons)$ if m is typed over TG and fulfills all constraints in $cons$.

Using this definition and the abstract syntax metamodel a modeling framework or tool is able to judge whether a model is consistent by checking whether the abstract syntax of the model is correctly typed and fulfills all constraints. Both can be checked automatically.

In order to define inter-model consistency we need a way to interrelate the abstract syntaxes of two or more models. For this purpose we define a composite model as a model that is composed of a set of other models. This is illustrated in Figure 2.

This figure shows a composite model in abstract syntax for two models A and B. The composite model contains both models. In compliance with our definition the abstract syntax of a composite model consists of an attributed typed graph. The containment relationship on the level of the abstract syntax denotes that the composite model contains the abstract syntaxes of the contained models as subgraphs. A formal definition of this relation can be found in Definition 3.

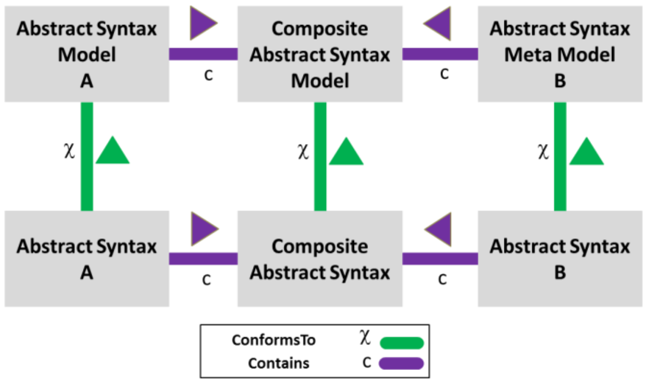

Figure 2. A composite model of two models A and B.

Definition 3. A model m in abstract syntax is a composite abstract syntax of $n \in \mathbb{N}$ models in abstract syntax $m_i, i \epsilon \{1, \dots, n\}$ if $\bigcup_{i=1,\dots,n}^{+} m_i \subset m$

The composite abstract syntax can also contain additional nodes and edges. These elements can be used to interrelate elements in the contained models and describe their relation.

Since the composite abstract syntax contains the union of the contained abstract syntaxes its type graph also has to contain a union of their type graphs. The relation between the composite ASMM and the ASMMs of the contained models is a containment relationship. Since the ASMM of the composite model is defined in compliance with Definition 1 this relation is defined as a component-wise containment of the type graphs and constraints. A formal definition of the composite abstract syntax metamodel is given in Definition 4.

Definition 4. An abstract syntax metamodel $asmm = (TG, cons)$ is a composite abstract syntax metamodel of $n \in \mathbb{N}$ abstract syntax metamodels $asmm_i = (TG_i, cons_i), i \in \{1, \dots, n\}$ if $\bigcup_{i=1,\dots,n}^{+} TG_i \subset TG$ and $\bigcup_{i=1,\dots,n}^{+} cons_i \subset cons$.

The type graph and constraints in the composite ASMM can contain additional elements. The type graph can contain additional node and edge types. They can be used to interrelate elements in the contained type graphs. The set of constraints can contain additional constraints. These constraints can be used to specify consistency requirements between the contained models that cannot be expressed on the layer of the individual models.

Since the composite abstract syntax metamodel still conforms to Definition 1 it is possible to define inter-model consistency of a set of models as intra-model consistency in their composite model. The definition can be found in Definition 5.

Definition 5. A set of n models $m_i, i \epsilon \{1, \dots, n\}$ in abstract syntax that are contained in a composite abstract syntax m are inter-model consistent if m is intra-model consistent.

Since inter-model consistency is a special case of intra-model consistency it can be validated with the same mechanisms.

The next section shows how the abstract syntax of two models can be described as AT-Graphs and how intra- and inter-model consistency can be expressed.

EXAMPLE

This section introduces an example for the described formalisms. For this example we assume a simplified runtime interpreter which interprets two models. The interpretation results in a software application.

The two interpreted models are a user interface model (UI model) and a state model. The UI-Model describes all windows that belong to this application as well as their layout and user interface elements. The state model is described as a finite state machine (FSM) and defines the states the application can be in. Each window in the user interface model is associated to one state. Whenever the state becomes active this window is shown. The transitions between states are associated to user interface elements. The transition fires whenever the user performs a certain action on the corresponding user interface element.

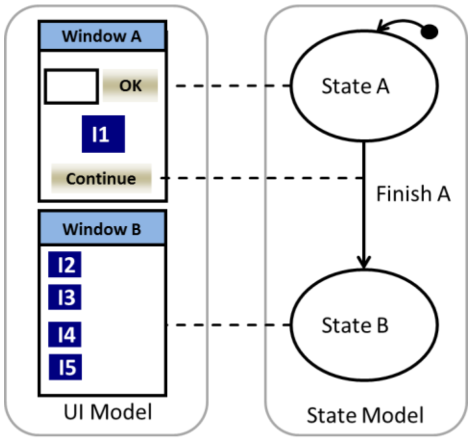

Figure 3. Both models and their connection in concrete syntax.

Figure 3 shows an example of both models in concrete syntax. The application in this example contains two states A and B. Accordingly the state model contains two states and a transition between them and the UI model contains two windows, one for each state.

The interrelations between both models are indicated by dashed lines. Each window is related to its respective state. The button labeled “continue” is connected to the transition from $State\ A$ to $State\ B$. Whenever this button is clicked the transition is fired. Accordingly $Window\ A$ is closed and $Window\ B$ is opened when $State\ B$ is reached.

In order for the framework to work correctly several consistency requirements have to be fulfilled. On one hand the used UI model and state model have to conform to the correct metamodel. For instance an unknown user interface element in the UI model cannot be interpreted by the framework. According to this, both models have to be intra-model consistent.

In addition, the following inter-model consistency requirements have to be fulfilled:

1. Each state has to be associated to a window. A state without a window can cause problems. If such a state is reached the interpreter cannot show any user interface to the user and consequently there are no user interface elements that could trigger a state change. Thus the application is in a dead-lock.
2. The user interface element associated to a transition has to be contained in the window that is associated to the origin state of this transition. If this is not the case the UI element is not shown when the transition is enabled. This means the transition can never fire.

The remainder of this chapter introduces the abstract syntax of both models and their composite model and shows how these consistency requirements are modeled.

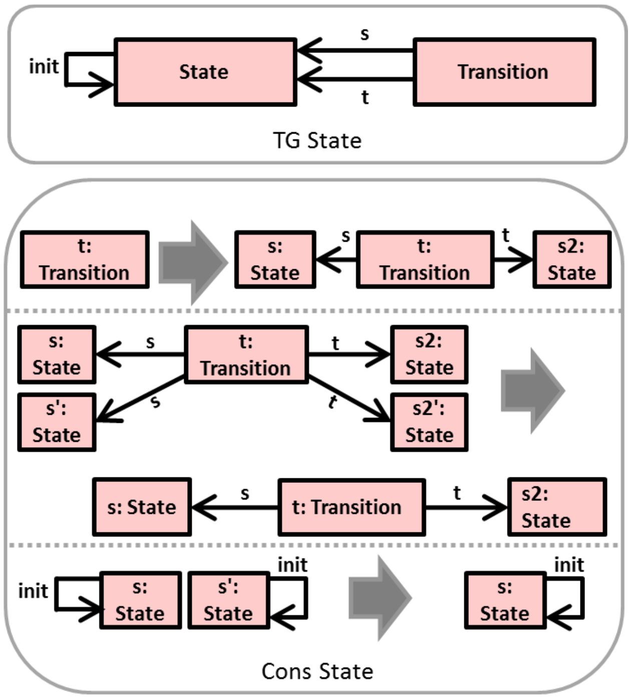

Figure 4. The abstract syntax metamodel of the state model.

Figure 4 shows the abstract syntax metamodel of the state model. It consists of the type graph $TG\ State$ and a set of constraints $Cons\ State$. For the purpose of this paper we focus on a limited version of the type graph of a FSM. This type graph contains the node types: $State$ and $Transition$ and edges denoting which states are the source (s) and target (t) of a transition. In addition an edge $init$ marks the initial state. Figure 4 also shows three of the constraints of this model. These first two constraints specify a transition to contain exactly one source and target. The first constraint can be read as: "if a transition exists it is always connected to at least one source and one target state". The second constraint specifies that a transition cannot have more than one source and target state. The third constraint specifies that only one initial state can exist.

Figure 5 shows an excerpt from the abstract syntax metamodel of the UI model. The type graph contains the types $Window$ and $UI\ Element$. The type $UI\ Element$ inherits several specific UI elements which are indicated in this figure. Amongst others, there are layout containers, labels and buttons. The window is connected to all of its UI elements via the edge type $ui\ elem$. The edge type $root$ identifies the root layout of the window. The sample constraints in Figure 5 state that every window is connected to exactly one root element.

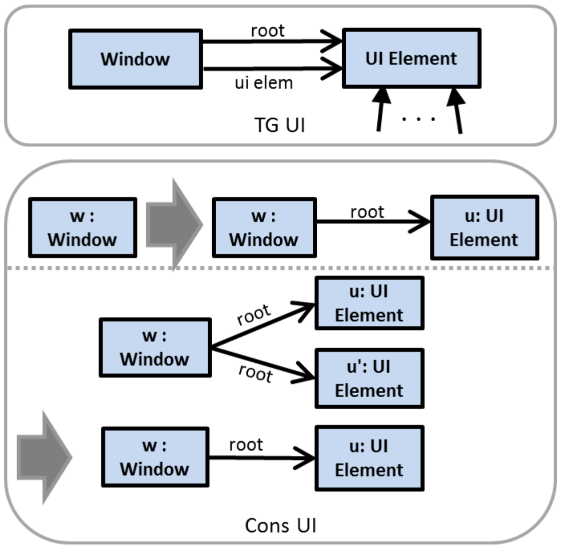

Figure 5. The abstract syntax metamodel of the UI model.

The inter-model consistency requirements are reflected in the composite model. Figure 6 shows the abstract syntax metamodel of the composite model. The type graph of this metamodel contains the type graph of the state model (upper part) and the type graph of the UI model (lower part). In addition it contains the type $State\,2\,Window$ which establishes the relation between the state and its window and the type $Transition\,2\,UI$ which denotes which UI element causes the transition to fire.

The constraints in the composite ASMM are responsible for ensuring that inter-model consistency requirements 1 and 2 hold. The first constraint can be read as "if there is a $State$, it has to be connected to a $Window$ via a $State\,2\,Window$ element". This ensures requirement 1. The second constraint can be read as "if a $UI\ Element$ is a trigger for a transition, this $UI\ Element$ has to be contained in the $Window$ that is associated to the source of this $Transition$". This ensures requirement 2.

The abstract syntax of our example in Figure 3 is an attributed typed graph over the type graph in Figure 6 that has to fulfill all constraints. This graph is depicted in Figure 7. It contains the abstract syntax of the state and UI model. The abstract syntax of the state model is depicted in the upper part of the composite abstract syntax. It consists of nodes for both states and the transition between them. The abstract syntax of the window model is depicted on the lower part of this figure. It contains two windows. The content of both windows is indicated by three dots. The composite abstract syntax also contains the additional elements that establish the relation between both models. Each state is connected to a window by a $State\ 2\ Window$ node. The transition is connected to the commit button in $Window$ A by a $Transition\ 2\ UI$ node.

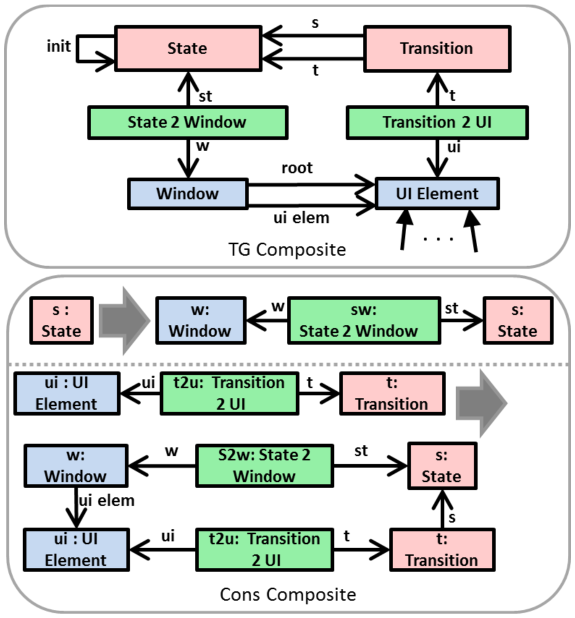

Figure 6. The abstract syntax metamodel of the composite model.

The model also fulfills the constraints, introduced in this section. The FSM contains one initial state and a transition that is connected to exactly one source and target state. Thus it fulfills the constraints from the ASMM of the state model. The abstract syntax of the state model is thus intra-model consistent with respect to its metamodel. Each window is also connected to exactly one root element (even if this is only indicated in the figure). Thus the UI model is intra-model consistent with respect to its ASMM

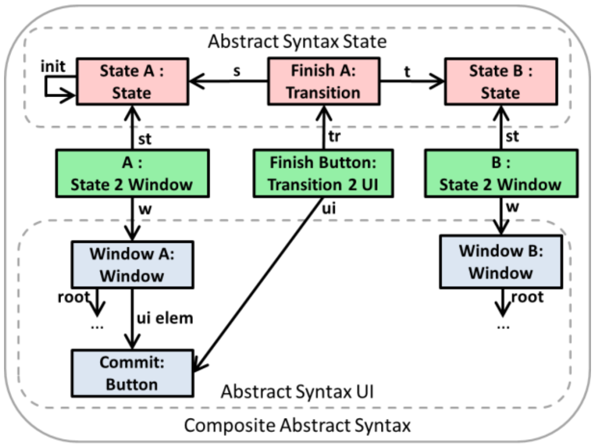

Figure 7. The abstract syntax of the composite model.

Both inter-model consistency requirements are also fulfilled. Each state is connected to a window and the commit button that triggers the transition $Finish\ A$ is contained in $Window\ A$ which is connected to the source state of this transition. Thus the model in Figure 7 is intra-model consistent with respect to the ASMM in Figure 6. Thus the contained state and UI model are inter-model consistent.

RELATED WORK

With the wide utilization of UML with its different models and the advent of domain specific languages for model engineering, intra- and inter-model consistency have been widely discussed.

Triple Graph Grammars, as introduced by Schürr [6], are a widely researched formal way for model to model transformation. For instance, Brandt et al. describe the abstract syntax of models as AT-Graphs [7]. In this approach the abstract syntaxes are interrelated by using a Triple Graph Grammar that describes how they can be constructed jointly. This approach is appealing for the joint construction of models and enables to generate one interrelated model from the other. In order to check for consistency both models have to be parsed with the grammar. The theory of Triple Graph Grammars still has to be extended to interrelate more than two graphs.

Usman et al. conveyed a Survey regarding different techniques for consistency checking between UML models [8]. The approaches are classified regarding the intermediate representation the consistency is checked on. The presented approaches are only able to interrelate special kinds of UML models and have not yet been applied as a general way to represent model consistency. The survey paper also introduces several consistency types and analysis parameters which can be a good starting point for future work.

A classification of the types of relationships between models participating in a software development process and possible inconsistencies is presented in [9]. Based on this classification, a set of requirements for generic inconsistency detection and a reconciliation mechanism is derived, which is suitable to serve as basis for future work on our formalism.

The idea of consistency checking via merges was first explored by Easterbrook and Chechik and applied in the framework for merging and reasoning about multiple, inconsistent state machine models [10]. Based on this approach, [11] aims at checking system consistency with respect to defined requirements. As requirements are addressed by different models, inconsistencies might be missed and it is difficult to take the heterogeneous (i.e. expressed in different languages) specifications into account. Thus it is proposed to translate heterogeneous requirements into model fragments which are instances of a common metamodel and to use model composition to merge these fragments into one unique model. Recent work of Sabetzadeh et al. [12] discusses model consistency checking through model merging. They present a technique for global consistency checking, supported by the TReMer+ tool [13] and identify patterns for consistency rules in conceptual modeling. Similar to our approach models are introduced as graphs and inter-model consistency is ensured through model merging. Lacking an abstract or, as they call it, logical model the approach cannot handle heterogeneous models yet. Additionally, it also focuses on conceptual and not so much formal models.

Finally, [14] proposes an alternative approach to model consistency by representing models by sequences of elementary construction operations. This allows the expression of structural and methodological consistency rules as logical constraints on these sequences. While aiming at being meta-model independent, the approach requires the definition of the constructional operations for each language.

CONCLUSION

This paper introduces a formalism for describing the abstract syntax of models as attributed typed graphs. Based on this formalism it is possible to describe consistency within models and between models in a homogenous way. This enables automatic validation of both types of consistency. Based on the introduced formalisms modeling tools and frameworks are able to check the contained models for consistency in order to spot possible problems early.

This paper also indicates the application of this notion of consistency in a small example. In the future, we plan to practically apply the approach to ensure consistency for adaptive user interfaces, generated at runtime on basis of a set of models.

Furthermore, we intend to work on extensions of this theory. There are additional formalisms like inheritance edges in type graphs and nested graph constraints that can considerably add to the expression power of the introduced formalism. Triple Graphs are also an appealing starting point for an alternative to our theory that will be explored in the near future. Using this technique it is possible to interrelate models without having to merge them. In this area we plan to explore the use of the Triple Graph formalism but slightly defer from the traditional theory of Triple Graph Grammars.

REFERENCES

1. Favre, J. M., Towards a basic theory to model model driven engineering, in *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*(2004)
2. Hermann, F., Ehrig, H. and Taentzer, G. A typed attributed graph grammar with inheritance for the abstract syntax of uml class and sequence diagrams, in *Electron. Notes Theor. Comput. Sci. 211* (Amsterdam, April 2008), Elsevier Science Publishers B. V., 261-269.
3. Limbourg, Q., Vanderdonckt, J, Transformational Development of User Interfaces with Graph Transformations, in *CADUI* (2005), Kluwer, 104-118
4. Ehrig, H, Prange, U, Taentzer, G., Fundamental Theory for Typed Attributed Graph Transformation, in *ICGT* (Berlin, 2004), Springer berlin / Heidelberg, 161-177
5. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K., Theory of Constraints and Application Conditions: From Graphs to High-Level Structures, in *Fundam. Inf. 74* (Amsterdam, 2006), IOS Press 135-166
6. Schürr, A., Specification of Graph Translators with Triple Graph Grammars, in *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science* (London, 1995), Springer-Verlag, 151-163
7. Brandt, C., Hermann, F., How Far Can Enterprise Modeling for Banking Be Supported by Graph Transformation?, in *Graph Transformations* (Berlin, 2010), Springer Berlin / Heidelberg, 3-26
8. Usman, M., Nadeem, A., Kim, T., Cho, E., A Survey of Consistency Checking Techniques for UML Models, *in Advanced Software Engineering and Its Applications* (Los Alamitos, 2008), IEEE Computer Society, 57-62
9. Easterbrook, S., Chechik, M., A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints, in *In Proceedings of International Conference on Software Engineering* (2001), IEEE Computer Society Press, 411-420
10. Perrouin, G., Brottler, E., Baudry, B., Le Traon, Y., Composing Models for Detecting Inconsistencies: A Requirements Engineering Perspective, in *Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ2009)* (Amsterdam, 2009), Springer Lecture Notes in Computer Science (LNCS)
11. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., Chechik, M., Consistency Checking of Conceptual Models via Model Merging, in *15th IEEE International Requirements Engineering Conference (RE 2007)* (2007), IEEE, 221-230
12. Sabetzadeh, M., Nejati, S., Easterbrook, S., Chechik, M., Global Consistency Checking of Distributed Models with TReMer, in *In 30th International Conference on Software Engineering (ICSE'08)* (2008)
13. Mao, X., Shan, L., Zhu, H., Wang, J., An adaptive casteship mechanism for developing multi-agent systems, in *Int. J. Comput. Appl. Technol.*(Geneva, 2008), Inderscience Publishers, 17-34
14. Blanc, X., Mounier, I. Mougenot, A., Mens, T., Detecting model inconsistency through operation-based model construction, in *Proceedings of the 30th international conference on Software engineering* (Leipzig, 2008), ACM, 511-520