Towards Verified Model Transformations

Holger Giese[1], Sabine Glesner[2], Johannes Leitner[3],
Wilhelm Schäfer[1], and Robert Wagner[1]

[1] Department of Computer Science
University of Paderborn, D-33098 Paderborn, Germany
[hg|wilhelm|wagner]@uni-paderborn.de
[2] Faculty IV – Electrical Engineering and Computer Science
Technical University of Berlin, D-10587 Berlin, Germany
glesner@cs.tu-berlin.de
[3] Department for Computer and Information Science
University of Konstanz, D-78457 Konstanz
leitner@inf.uni-konstanz.de

Abstract. Model-driven software development (MDD) is seen as a promising approach to improve software quality and reduce production costs significantly. However, one of the problems in using MDD especially in the area of safety-critical systems is the lack of verified transformations. The verification of crucial safety properties on the model level is only really useful, if the automatic code generation is also guaranteed to be correct, i.e., the verified properties are guaranteed to hold also for the generated code. This particularly means to check semantic equivalence, at least to a certain extent between the model specification and the generated code. This paper addresses the problem of verifying that a given transformation ensures semantic equivalence between an arbitrary model in a given model specification language and the resulting programming language code. While the presented approach ensures that the transformation algorithm is correct, existing related work is restricted on verifying only the correctness of a particular transformation result.

1 Introduction

Model-driven software development (MDD) is seen as a promising approach to improve software quality and reduce production costs significantly. A major basis of such an approach is a usually domain-oriented modeling language which enables to abstract from implementation specific details and thus makes models (much) easier to develop and analyze than the final implementation. A significant additional benefit in terms of improved quality and reduced costs could be gained by the fully automatic transformation of a model-based system specification into executable code, if at all possible.

In developing safety-critical systems this approach is getting increasing attention, as model analysis has advantages over pure testing of implemented systems. Important required safety properties of a system under development could be verified on the model level rather than trying to systematically test the absence of failures. Prominent failures in the past illustrate that testing often fails to detect malfunctioning by overlooking particular scenarios.

However, one of the problems in using MDD especially in the area of safety-critical systems is the lack of verified transformations. The verification of crucial safety properties on the model level is only really useful, if the automatic code generation is also guaranteed to be correct, i.e., the verified properties are guaranteed to hold also for the generated code. This particularly means to check semantic equivalence, at least to a certain extent between the model specification and the generated code.

While testing model transformations [1] and in particular approaches which exploit the specification of the code generator to derive critical test cases [2] are a valuable aid to ensure the quality of the transformation, they can only check a finite number of cases and thus fail to ensure the required semantic equivalence.

This paper addresses the problem of formally verifying that a given transformation ensures semantic equivalence between any model of the given model specification language and the resulting programming language code.

In compiler construction several approaches exist which check correctness of a transformation algorithm in particular or the correctness of the implementation when going from the code to lower level code or executables, see [3] for an overview. As a detailed example, on the level of source code transformations, Java and its transformation in Java byte code have been extensively investigated [4, 5]. The approach of proof-carrying code [6] is also weaker than what we intend to provide, because it concentrates only on the verification of necessary but not sufficient correctness criteria. The approach of program checking has been proposed by the Verifix project [7] and has also become known as translation validation [8, 9], recently also for loop transformations [10]. For an overview and for results on program checking in optimizing backend transformations cf. [11].

In contrast to these approaches for compiler construction, model to code transformations are characterized by rules and pattern matching like activation schemes of these rules, and thus the techniques employed in compiler construction are not directly applicable for their formal verification.

Although there exist many approaches for the specification and execution of model transformations, to the best of our knowledge, the only approaches addressing the problem of semantic equivalence in the above sense, at least to a limited extent, consider specific model instances and their translation results. Either specific correctness conditions are checked for both the original model and its transformation [12] or the semantic equivalence between both models is guaranteed by a bisimulation check [13].

Our approach realized in the Fujaba Tool Suite[4] is based on a formal specification technique, namely triple graph grammars (TGG) to specify a model to code transformation. The correctness of this specification and consequently the code generator is shown using the theorem prover Isabelle/HOL.

The next section will illustrate the use of a domain-oriented modelling language based on the example of a production line and its fairly complex control

[4] www.fujaba.de

software. This realistic example of an existing industrial system will also be used to present the TGG and their application in building model to code transformations in Section 3. Section 4 describes the use of ISABELLE/HOL to verify the transformation. Section 5 gives an account of the status of our work by listing required next steps and remaining open questions. Finally, Section 6 summarizes the work.

2 Modeling Approach

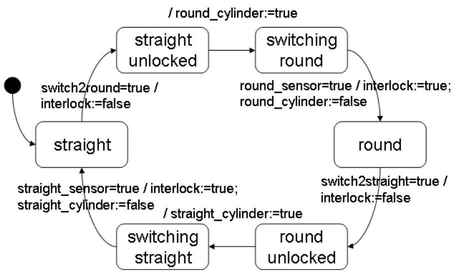
In this section, we provide a brief overview of the employed modeling approach [14] using a simple case study from the area of flexible production systems. It exemplifies the need for dependable model transformations and serves as a running example for explaining the specification of model transformations and their verification.

The substantial components of this modular system are working stations, straight and curved monorail tracks, as well as transfer gates (switches). For the transportation of materials and goods between the working stations self-propelled transportation units (shuttles) moving along the tracks are employed. The transportation units circulate on the main loop of the material flow system and can be stopped at stations or before curves and transfer gates only.

The decentralized production control system consists of PCs on the supervisory control level and Programmable Logic Controllers (PLC) on the cell level. The components' actuators and sensors are connected to the PLC via an Actuator Sensor Interface (ASI). The communication among PCs and PLCs is implemented by a multi-point interface (MPI, Siemens AG). Higher-level tasks, e.g., planning, order assignment, and coordination of local activities of all controllers are done at the supervisory control level. The PLCs on the cell level are responsible for the control of local components such as stations or transfer gates.

For the specification of the control software, we combined subsets of the Specification and Description Language (SDL) [15] and the Unified Modeling Language (UML) [16] into an executable graphical language [17]. In this language, a block diagram is used to specify the overall static communication structure where processes and blocks are connected to each other by channels and signal routes. For implementation purposes, the block diagram is automatically transformed to an initial class diagram. This class diagram can be refined and extended to an executable specification. For example, we can assign an automaton to model the reactive behavior of the control software. Fig. 1(a) presents a simple automaton for the control of the transfer gate used in our case study.

The specified automaton switches the transfer gate between the straight and the round direction. Initially, the transfer gate is switched to the straight direction and fixed by a mechanical interlock. When the condition *switch2round=true* becomes true, the interlock is disengaged by the action *interlock:=false*, and state *straight unlocked* is entered. Thereafter, the triggerless transition fires and the appropriate action *round_cylinder:=true* is executed. This action activates the pneumatic cylinder responsible for turning the transfer gate into the round di-



(a) Transfer gate automaton

```
VAR
  state : INT := 1; /* set to 'straight' */
END_VAR;

CASE state OF

  1: /* current state is 'straight' */
   IF switch2round=true THEN
     interlock:=false;
     state:= 2; /* set to 'straight unlocked' */
   END_IF;

  2: /* current state is 'straight unlocked' */
   round_cylinder:=true;
   state:=3; /* set to 'switching round' */

  ...

END_CASE;
```

(b) Generated PLC-code

Fig. 1. Automaton and generated PLC-code controlling a transfer gate

rection. The state *switching round* is left if the proximity sensor announces that the switching process completed. If the switching process was successful, the state *round* is entered and the interlock re-engaged. Now, the transfer gate can be switched back to the straight direction which is performed analogous to the described switching process for the round direction.

For the specification of the entire system, further controller automata are needed, e.g., automata to control the stopping and starting of shuttles at stations or before transfer gates. The sheer number of these automata and their interaction makes it hard to check manually whether the system functionality is defined correctly. As an example consider a requirement like "a shuttle never enters a transfer gate if the transfer gate is currently switching its direction". Our approach enables automated verification of such kind of safety-critical requirements using model checking [18]. After a successful verification, the controller automata need to be implemented. In our approach, the defined precise semantics of the automata model allows us to generate the PLC-code automatically.

PLCs are microprocessor systems that are widely used in industrial automation. The reason for their popularity is that they are robust and reliable. A PLC is connected to sensors and actuators: the former provide information on the state of the controlled component while the latter perform the actions prescribed by the control software. PLCs behave in a cyclic manner where each cycle follows three phases: (1) poll all inputs and store read values, (2) compute new output values, and (3) update all outputs. The repeated execution of this cycle is managed by the built-in real-time operating system. Thus, the control software has to compute the output values based on the read input values only.

For the automatic generation of PLC-code out of an object-oriented specification, we adapted our code generation mechanisms to produce Structured Text (ST). Structured Text is a notation similar to PASCAL. It provides constructs such as if-then-else-conditionals and while-loops. As typical object-oriented concepts like inheritance or polymorphism are not supported, we implement the behavior of an automaton by simple switch-case constructs. The piece of code in Fig. 1(b) is an excerpt of the generated PLC-code for the automaton shown in Fig. 1(a) and gives a short impression on the translation in Structured Text for the states *straight* and *straight unlock*.

First, each state of the automaton is assigned a unique integer value. Then, we declare an integer variable *state* to keep the current state of the automaton which is handled in a case-statement. For our example automaton, the current state variable *state* is set to the initial state *straight* represented by the assigned integer value.

The outgoing transitions are encoded as if-statements with the transition guard as condition. For triggerless transitions, the if-statement is omitted if it is the one and only outgoing transition from that state. If they are more outgoing transitions with a guard, the triggerless transition is embedded in a if-else statement. Multiple triggerless transitions from one state are forbidden. The actions are realized as simple variable assignments. These variables, together with the variables from the conditions, are mapped by the compiler to the real addresses of the hardware. Note that the presented program is executed once in each cycle of the PLC. Thus, it is the body of an implicit loop-forever statement.

3 Model Transformations

To realize the modeling approach outlined in the previous section, we need a transformation which translates the given automata into executable PLC-code. Since code can be also viewed as a more detailed model of the software, we employ for this translation a model transformation technique based on triple graph grammars [19]. In this section, we give an overview of our model transformation approach and introduce the basics of the employed model transformation technique using our example from the previous section.

Fig. 2 gives an overview of our model transformation approach. The model transformation is specified by a number of transformation rules. The transformation rules are specified w.r.t. the metamodels of the source, the target, and an additional correspondence metamodel. From these rule specifications, a transformation engine is generated. The automatically derived engine transforms a source model into a target model yielding an additional correspondence model. This correspondence model enables a clear distinction between the source and the target model and holds additional traceability information about the applied mappings between the involved model elements. This information is used for further incremental updates if one of the models changes [20]. Hence, after an initial transformation the correspondence model serves as an additional input for following update transformations. In addition, since the employed transformation technique is bidirectional in nature, the source and target models can change their roles and a reverse transformation, i.e., form the target to the source model, will be also possible. However, to keep things simple, in this paper we consider only transformations in the forward direction, i.e., from the source to the target model.

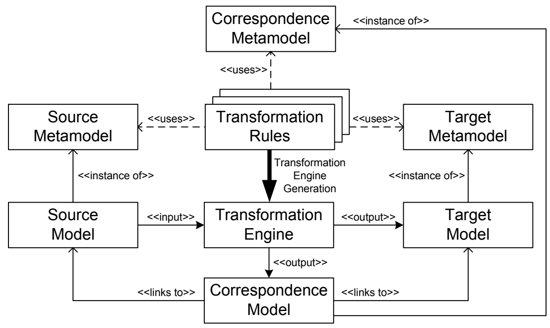


Fig. 2. Overview of the model transformation approach

In order to explain the specification technique of triple graph grammars for model transformation, we have to take a closer look at the involved metamodels. A metamodel defines the abstract syntax and static semantics of a modeling language. In Fig. 3, the automata metamodel, the metamodel defining the abstract syntax tree of the Structured Text programming language for PLCs, and the correspondence metamodel are shown.

In the automata metamodel shown in the upper left of Fig. 3, an *Automaton* consists of *States* and *Transitions*. A *Transition* connects *States* by its *outgoing* and *incoming* associations and has a *Trigger* as well as an ordered sequence of *Actions*. Some special states are the classes *InitialState* and *FinalState*. An automaton can have only one *InitialState* referenced by the directed association *initialState* but many *FinalStates* though they are rarly used for the specification of reactive systems.

For the specification of a triple graph grammar, we need an additional correspondence metamodel. It is shown in the upper right of Fig. 3. The metamodel defines the mapping between a source and a target metamodel by the classes *TGGNode* and *Object* and its associations *sources* and *targets*. Since all classes inherit implicitly from the *Object* class (not shown here), the correspondence model stores the traceability information needed to preserve the consistency between two models. In addition, the class *TGGNode* has a self-association *succ* which connects the correspondence nodes with their successor correspondence nodes. This extra link is used by our transformation algorithm.

The two described classes and their associations are essential for our transformation algorithm. However, further correspondence classes and refined associations can be added. In our example, we have added two additional correspondence classes, including the correspondence class *CorrNode* used in our example rule (cf. Fig. 4). The additional correspondence classes increase the performance of our transformation algorithm but have no impact on the carried out formal verification.

The abstract syntax tree for Structured Text is defined by the metamodel shown in the lower part of Fig. 3. In fact, we are using only a subset of the

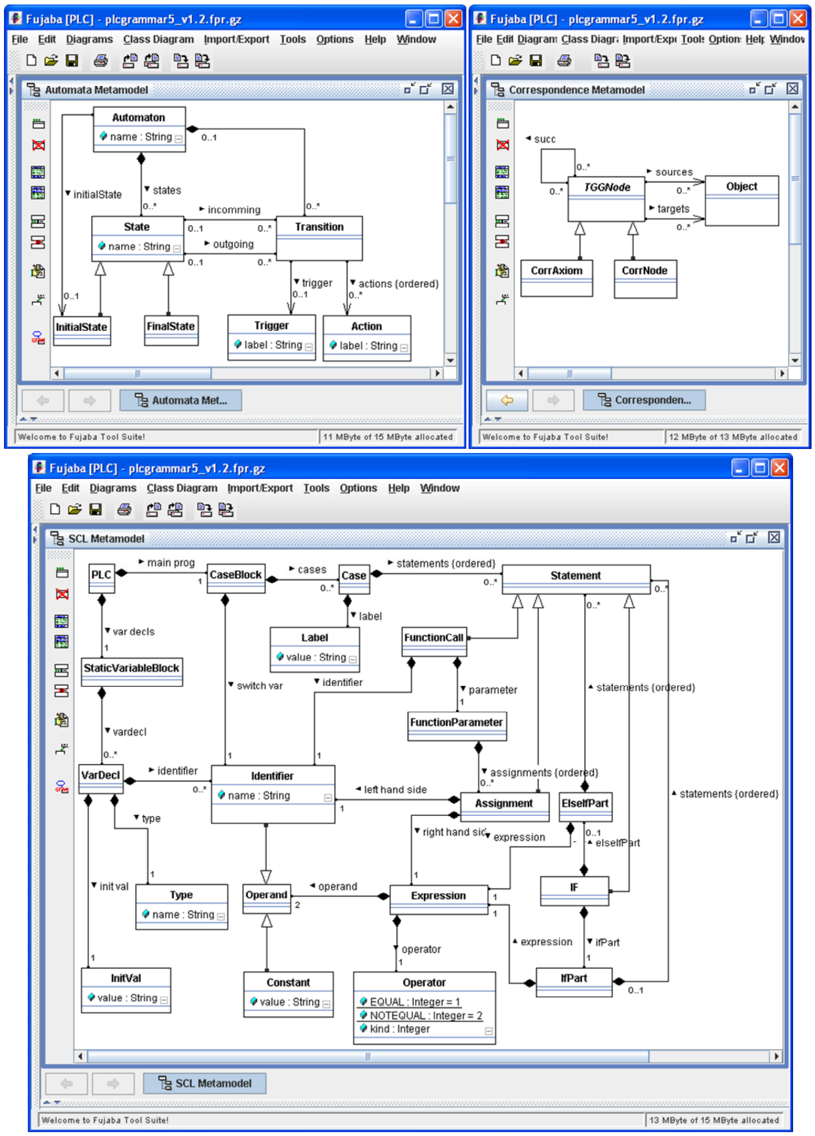


Fig. 3. Metamodels of the source, correspondence, and target model

language that is needed for the code generated out of automata. This subset was extracted from the Structured Text grammar definition and comprises basically case-switch statements, if-then-else statements, assignment statements as well as expressions.

A program is represented by the class *PLC* that consists of one *StaticVariableBlock* and a *CaseBlock*. The class *StaticVariableBlock* has a to-many composition association to the class *VarDecl* which represents a variable declaration. A variable declaration comprises a *Type*, an *Identifier*, and an *InitVal* class representing the initial value of the identifier. The *CaseBlock* relates to an *Identifier* and is associated to many *Cases* that are represented by a *Label*. Each *Case* comprises a sequence of ordered *Statements*. A *Statement* is either a *FunctionCall* with a *FunctionParameter* whose result is assigned to an *Identifier*, an *Assignment* with a left-hand side *Identifier* and a right-hand side *Expression*, or an *IF* condition block. The *Expression* is defined by two *Operands* and an *Operator*. Up to now, only two kinds of operators are supported: equality and inequality. The *Operands* can be also represented by an *Identifier* or a *Constant*. An *IF* condition consists of an *IfPart* and an optional *ElseIfPart* which both have an *Expression* and embody an ordered sequence of *Statements*.

Given these three metamodels, a triple graph grammar for our example model transformation can be specified. In the following, we will explain the basic concepts with the help of our example and refer to [19] for a formal definition.

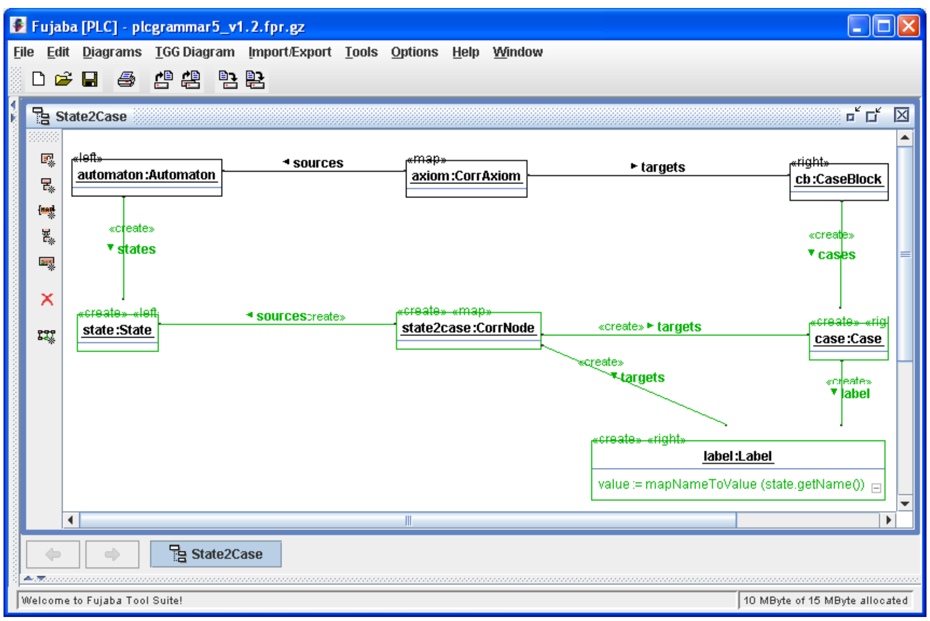


Fig. 4. A triple graph grammar rule mapping states to case statements

A triple graph grammar specification is a declarative definition of a bidirectional model transformation. In Fig. 4, a triple graph grammar rule in the FUJABA-notation is depicted. The rule specifies a consistent correspondence mapping between the objects of the source and the target model. In particular, the presented rule defines a mapping between a state and a corresponding case statement. The objects of the automaton are drawn on the left and the objects of the programming language are drawn on the right. They are marked with the ≪left≫ and ≪right≫ stereotypes respectively. The correspondence objects in the middle of the rule are tagged with the ≪map≫ stereotype.

The rule is separated into a triple of productions (source production, correspondence production, and target production), where each production is regarded as a context-sensitive graph grammar rule. A graph grammar rule consists of a left-hand side and a right-hand side. All objects which are not marked with the ≪create≫ stereotype belong to the left-hand side and to the right-hand side; the objects which are tagged with the ≪create≫ stereotype occur on the right-hand side only. In fact, these tags make up a production in Fujaba's graph grammar notation.

The source production on the left shows the generation of a new state and linking it to an automaton. The target production on the right shows the addition of a new case statement and its linking to the case block. In addition, the case block is equipped with a label to identify the state in the program. Since states in the program are encoded as integer values, a mapping function is used to translate the name of the state to a unique integer value. The correspondence production in the middle shows the relations between a state and the objects representing the case statement.

A graph grammar rule is applied by substituting the left-hand side with the right-hand side if the pattern of the left-hand side can be matched to a graph, i.e., if the left-hand side is matched all objects tagged with the ≪create≫ stereotype will be created. Hence, our example rule, in combination with additional rules covering other elements, can generate an automaton with the corresponding representation in the programming language by applying the production triples simultaneously. However, the transformation will not be executed this way. To execute a transformation, conceptually, we can assume that whenever a state is added to the automaton, a case statement with a corresponding label will be generated in the program. This way, the triple graph grammar rules define a transformation between automata and their representation in the programming language Structured Text.

The briefly described model transformation approach was realized in the Fujaba Tool Suite. For the visual specification of a triple graph grammar rule we use the TGGEditor (cf. Fig. 4) which is realized as a plug-in. This editor ensures conformance to the source, the correspondence, and the target metamodels. For this purpose, the required metamodels have to be specified in Fujaba as class diagrams (cf. Fig. 3).

The execution of a model transformation is done by the MoTE plug-in. MoTE is the abbreviation for Model Transformation Engine. It is the core library for the execution of triple graph grammars and can be also used without Fujaba. In order to execute a model transformation, we generate from each triple graph grammar rule Java code using Fujaba's code generation facilities. This code is compiled to executable transformation rules which are bundled into a single Jar archive file. The archive represents the catalog of transformation rules defining the model transformation specified by a triple graph grammar.

Once the catalog is available, the transformation engine is complete and model transformations can be carried out.

As mentioned in Section 2, our approach verifies the specified automata w.r.t. crucial safety requirements. However, the proven properties can only be guaranteed to hold also for the implementation, if we can ensure that the implementation realizes the same behavior as the specified automata. Therefore, we have to ensure that the employed model transformation from the automaton model to the code model is correct (the implementation model must be semantically equivalent to the already verified automata model).

4 Verification

In this section, we describe our approach for the verification of triple graph grammar transformations in ISABELLE/HOL. We show in more detail how to derive ISABELLE/HOL representations from structures in FUJABA and outline the basic proof scheme.

In essence, we prove that the relation of semantic equivalence is a congruence with respect to an appropriate representation of the transformation rules. Fig. 5 extends the modeling overview from Fig. 2 with an illustration of our general proof scheme.

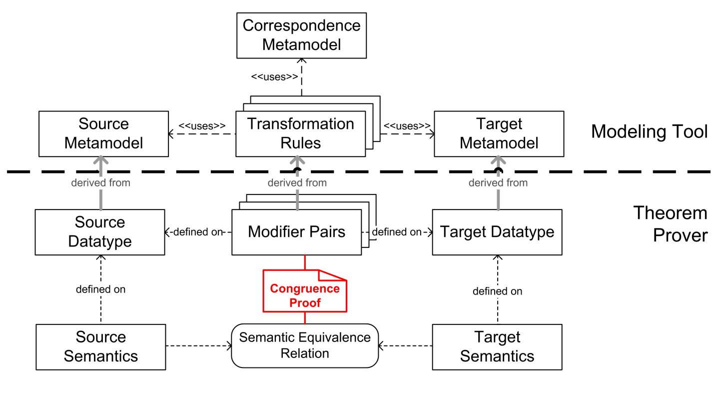


Fig. 5. Overview of the verified model transformation approach

Note that the model instances (as well as the transformation engine) shown in Fig. 2 are omitted here. Since we verify the correctness of transformation rules applied to *any* model of the specified type, these instances are irrelevant for the proof.

Model transformations are often formalized as instances of graph transformations. While this approach is intuitive and shifts the problem into an extensive and well-known theory, its realization in ISABELLE/HOL poses a number of difficulties. Problems already arise when trying to formalize models as instances of metamodels in HOL. A metamodel entails a number of structural constraints on its instances, while a graph just consists of arbitrarily connected nodes. Metamodel constraints have to be expressed as additional axioms about the structure of the graph, like "Nodes with a type of `State` can only be directly connected to nodes of type `Transition`". Even small metamodels will result in graph types encompassed by long lists of such axioms. Defining semantics and conducting proofs on these structures is tedious and, more importantly, the derivation of axioms from metamodels is not straight-forward, has to be done manually and is thus error-prone.

For these reasons, we chose a different formalization of metamodels that comprises all the structural information directly in a type definition. This makes proofs simpler and significantly more compact. At first, we create a modified version of the metamodel with an ordered, tree-like structure. This structure can always be achieved by, for example, converting circular compositions to reference attributes. Fig. 6 shows the result for a part of the metamodel of the employed automata presented in Fig. 3.

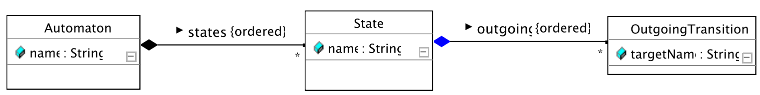


Fig. 6. Part of the modified metamodel for automata

This kind of "flattened" model can be mapped to a type in Isabelle/HOL using only constructs like records, lists and other primitive data types. The nature of this mapping is straight-forward and might be implemented as an automatic procedure in the future. The result for the above metamodel is:

```
record State =                                  record OutgoingTransition =
  Identity   :: BaseType                          Target  :: BaseType
  Outgoing   :: OutgoingTransition list           Actions :: ActionType list
  FinalState :: bool                              Trigger :: TriggerType option
```

Primitive (i.e. algebraic) types are a core concept of Isabelle/HOL. On these types, we can easily define an operational semantics as a recursive function over the structure of the model. On this semantics, we define a bisimulation $\approx$, formalizing the notion of *semantic equivalence*. In many cases, semantic equivalence is just defined as (statewise) equality; however, different semantic domains of source and target model might require a more abstract comparison. Our definition of semantics and semantic equivalence is shown in detail in [21, 22].

We view rules of a triple graph grammar not as specifications of transformations on a single graph, but as *pairs* of productions describing a way that two models are simultaneously modified. This allows for an easy and elegant formalization in Isabelle/HOL. Fig. 7 shows the pair of graph productions corresponding to the triple graph grammar rule that maps states to case statements (shown in Fig. 4).

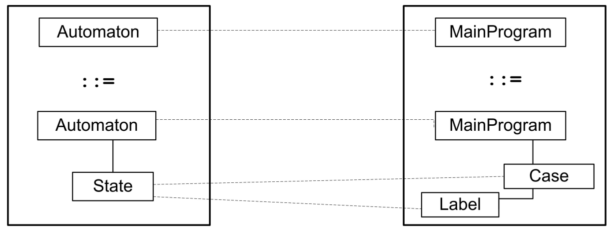


Fig. 7. View of the triple graph grammar rule in Fig. 4 as a pair of productions

This interpretation captures the bidirectional nature of TGGs by interpreting them as a grammar for the parallel evolution of source and target model. For each of these productions we can now define an operator, called *modifier*, on source and target model. We formalize the application of a transformation rule as a parallel application of the corresponding modifiers to both models. For example, for the productions above, we define modifiers for automata and PLC programs that add a state or a case statement, respectively:[5]

$$A \oplus s \equiv A(\!|\ \text{States} \ := (\text{States}\ A) \cdot s\ |\!)$$
$$P \oplus c \equiv P(\!|\ \text{MainProgram} \ := c \cdot (\text{MainProgram}\ P)\ |\!)$$

For the correctness of each transformation rule it then suffices to show that the application of the modifiers will not destroy semantical equivalence of models. For example, if we add a state to an automaton and a corresponding case block to a semantically equivalent PLC program in the Structured Text programming language, automaton and program remain equivalent:

$$A \approx P \implies (\ A \oplus s\ ) \ \approx \ (\ P \oplus \texttt{State2Case}(s)\ ) \tag{1}$$

In effect, we show that semantical equivalence is a congruence with respect to every transformation rule. The proof for the above lemma is straight-forward, since neither the new state nor the new code segment will be reachable. However, more complex rules require more elaborate proofs. For example, the proof for one variant of the TGG rule `Action2Code` that adds actions to transitions (and inserts PLC code in the appropriate location in the program) requires over 100 lines of proof code[6] in ISAR notation [23] and makes use of 15 additional helper lemmas. In total, the proof of the correctness of the transformation from automata to a PLC language contains approximately 1500 lines of proof code. Table 1 shows the distribution of lines of proof code for the different parts of the proof.

[5] Here, the Isabelle/HOL operator $(\!|... := ...|\!)$ is used to update the specified member of a record. The operator $\cdot$ appends elements to lists.
[6] Note that proofs in Isabelle/HOL cannot be done automatically and that the vast majority of proof steps needs manual interaction.

Table 1. Lines of ISAR proof code for different parts of the proof

Mapping		
	Automata formalization	170
	SCL formalization	259
	TGG rule formalization	302
Rule Correctness Proofs		
	Definition of semantic equivalence	42
	The `Axiom` rule	22
	The `State2Case` rule	81
	Two variants of `Transition2Code`	328
	Two variants of `Action2Code`	276

5 Next Steps

To realize the vision of MDD by means of verified model transformations, the presented results are a first step. We discuss in this section required next steps we plan to address.

Rule correctness lemmas of the simple form as presented in Section 4 will not always be provable. TGG rules are not applicable on arbitrary patterns in the source and target graph, but rely on the correspondence graph created by previous rules. For example, adding a transition to a state of an automaton and a corresponding code segment to a case-block of a PLC program will only preserve semantic equivalence if the state and the case-block themselves correspond to each other, i.e., were created by a pair of modifiers. In fact, in the example lemma (1) the correspondence is hidden in the Function `State2Case`, which creates a case block with the same identifier as the state s. This results in additional preconditions, which we call *correspondence preconditions*, for the rule correctness lemmas. At the moment, we tackle this problem by introducing the correspondence preconditions manually. We aim to show that these preconditions indeed result from previous rule applications. A possible solution makes use of the set of all models introduced by successive application of all the rules in a TGG grammar, which would enable us to conduct proofs about the relationships between rule applications.

In addition to a proof technique, also the methodological aspects of verifying model transformation have to be addressed. We therefore plan to elaborate the design and verification process for model transformations and develop automated or semi-automated tool support for the required activities where possible.

A first planned extension is to automatically derive the formalization of the metamodels and TGG rules in ISABELLE/HOL which accounts for nearly 50% of the lines of proof code. We also want to explore how we can combine the interactive theorem proving with available automated verification approaches for finite and infinite graph transformations already present in FUJABA [24] in order to reduce the effort for the verification of a model transformation.

6 Conclusions

Model-driven software development, especially with domain-specific languages, is increasingly important to automatically develop software that adheres to its specification. In this paper, we have shown how model-driven software development is applied in the context of flexible production systems. These systems and their transformations are specified within the FUJABA TOOL SUITE using triple graph grammars (TGGs). TGGs are a special form of graph grammars that allow us to specify the parallel evolution of systems, namely of the source (or model) system and the target system (its implementation). We have presented results of ongoing work how such transformations can be formalized and verified in the ISABELLE/HOL theorem prover. This is an important step towards fully verified model transformations, which are necessary to guarantee correctness of the generated implementations of the specified models.

References

1. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: Testing model transformations. In: Proc. of the First International Workshop on Model, Design and Validation, Rennes, November 2004. (2004) 29–40
2. Stuermer, I.: A contribution of graph grammar techniques to the specification, verification and certification of code generation tools. Electronic Notes in Theoretical Computer Science **72** (2002) 10
3. Glesner, S., Goos, G., Zimmermann, W.: Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). it - Information Technology **46** (2004) 265–276 Print ISSN: 1611-2776.
4. Klein, G., Nipkow, T.: Verified Bytecode Verifiers. Theoretical Computer Science **298** (2003) 583–626
5. Klein, G., Strecker, M.: Verified Bytecode Verification and Type-Certifying Compilation. The Journal of Logic and Algebraic Programming **58** (2004) 27–60
6. Necula, G.C.: Proof-Carrying Code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France (1997) 106–119
7. Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., von Henke, F., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H., Zimmermann, W.: Compiler Correctness and Implementation Verification: The Verifix Approach. In Fritzson, P., ed.: Poster Session of CC'96, IDA Technical Report LiTH-IDA-R-96-12, Linkoeping, Sweden (1996)
8. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In Steffen, B., ed.: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, Springer Verlag, LNCS 1384 (1998) 151–166
9. Necula, G.C.: Translation Validation for an Optimizing Compiler. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), Vancouver, British Columbia, Canada (2000) 83–94
10. Goldberg, B., Zuck, L., Barrett, C.: Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers. In: Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2004), 7th European Conferences on Theory and Practice of Software (ETAPS 2004), Barcelona, Spain, Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS) (2004)

11. Glesner, S.: Using Program Checking to Ensure the Correctness of Compiler Implementations. Journal of Universal Computer Science (J.UCS) **9** (2003) 191–222
12. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In Jürjens, J., Rumpe, B., France, R., Fernandez, E.B., eds.: CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop. Number TUM-I0323 in Technical Report, Technische Universität München (2003) 63–78
13. Narayanan, A., Karsai, G.: Towards verifying model transformations. In: 5th International Workshop on Graph Transformations and Visual Modeling Techniques, Vienna, 2006. Electronic Notes in Theoretical Computer Sience (2006) 185–194
14. Schäfer, W., Wagner, R., Gausemeier, J., Eckes, R.: An engineer's workstation to support integrated development of flexible production control systems. In Ehrig, H., Damm, W., Desel, J., Groše-Rhode, M., Reif, W., Schnieder, E., Westkämper, E., eds.: Integration of Software Specification Techniques for Applications in Engineering. LNCS 3147. Springer Verlag (2004) 48–68
15. International Telecommunication Union (ITU), Geneva: ITU-T Recommendation Z.100: Specification and Description Language (SDL). (1994 + Addendum 1996)
16. OMG 250 First Avenue, Needham, MA 02494, USA: Unified Modeling Language Specification Version 1.5. (2005)
17. Nickel, U., Schäfer, W., Zündorf, A.: Integrative specification of distributed production control systems for flexible automated manufacturing. In Nagl, M., Westfechtel, B., eds.: DFG Workshop: Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen, Wiley-VCH Verlag GmbH and Co. KGaA (2003) 179–195
18. Giese, H., Kardos, M., Nickel, U.: Integrating Verification in a Design Process for Distributed Production Control Systems. In: Proceedings of the 2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT2002), Grenoble, France. (2002)
19. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graph-Theoretic Concepts in Computer Science, 20^{th} International Workshop, WG '94. Volume 903 of LNCS., Herrsching, Germany (1994) 151–163
20. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genoa, Italy. LNCS, Springer Verlag (2006).
21. Leitner, J.: Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken. Master's Thesis (*Diplomarbeit*), University of Karlsruhe (2006)
22. Blech, J.O., Glesner, S., Leitner, J.: Formal Verification of Java Code Generation from UML Models. In: Proceedings of the 3rd International Fujaba Days 2005: MDD in Practice, Technical Report, University of Paderborn (2005)
23. Nipkow, T.: Structured Proofs in Isar/HOL. In: Types for Proofs and Programs (TYPES 2002), Springer Verlag, LNCS 2646 (2003) 259–278
24. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proc. of the 28^{th} International Conference on Software Engineering (ICSE), Shanghai, China, ACM Press (2006)