Electronic Communications of the EASST
Volume 42 (2011)

Proceedings of the 4th International Workshop on Multi-Paradigm Modeling (MPM 2010)

Asserting the Correctness of Translations

Bruno F. Barroca, Vasco M. Amaral

12 pages

Guest Editors: Vasco Amaral, Hans Vangheluwe, Cécile Hardebolle, Lazlo Lengyel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Asserting the Correctness of Translations

Bruno F. Barroca[1], **Vasco M. Amaral**[2]

[1] Bruno.Barroca@di.fct.unl.pt, http://citi.di.fct.unl.pt/member/member_act.php?id=77
[2] Vasco.Amaral@di.fct.unl.pt, http://ctp.di.fct.unl.pt/~va/
Centro de Informática e Tecnologias de Informação (CITI)
Departamento de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, Portugal

Abstract: While building a new language, we assign its semantics by mapping its syntax onto a semantic domain. To do so, we can either *(i)* do it operationally, by means of small-step morphisms within the same semantic-domain; or *(ii)* by means of a translation (syntax-to-syntax transformation), onto a target language that has already an operational semantics defined. Despite the fact that it is possible to build the set of syntactic correspondences from a given translation, it is still not clear how we can assert about the correctness of these syntactic correspondences in w.r.t. both the source and target language's underlying semantics.

In this paper, we combine the above described techniques by analyzing the translation and establishing a semantic relation between the respective operational semantics, in order to assert the correctness of that translation. We demonstrate our approach with a concrete translation between two languages: State Machines and Petri Nets; and decide about its correctness by using their respective operational semantics as oracles. Finally, we discuss about the validity of our assertions in w.r.t. language translations in general.

Keywords: Translations, Model Transformations, Structured Operational Semantics (SOS), Translation Analysis, Translation Symbolic Space, Bissimulation-equivalence

1 Introduction

One of the challenges of multi-paradigm modeling is to be able to accept that we can look at any existing (or intended) artifact in multiple perspectives simultaneously, in a sound and coherent fashion — i.e without the existence of logical contradictions between the involved views over that artifact. These different views can be realized by means of domain specific modeling languages (DSMLs) [KT08], hence enabling the end-users — the modelers involved in a domain specific modeling (DSM) activity — a pragmatic and usable way to understand their particular views. These DSMLs, capture the occurrence of reusable patterns while describing an artifact in a particular perspective (i.e limited to a given domain), and provide to the end-users a pragmatic way to apply these reusable patterns in a controlled fashion.

The concern on model comprehension refers to the cognitive capabilities of the end-users while understanding their own models and dealing with its intrinsic complexity. Hence, the quality level of a model is specially biased to the computational complexity of the underlying

analysis algorithms involved in the extraction of a clear rational over the specified models. In general, depending on the expressiveness of the used language, the size of the model's analysis spaces can be unreasonable and reach the limits of undecidability [Plu98].

One solution to solve this complexity management problem while dealing with models is to provide syntax-to-syntax model transformations (or translations throughout this paper) between DSMLs. The translations can enable the reuse of the capabilities offered by the different languages and associated tools, such as efficient analysis algorithms and data structures, simulation and visualization capabilities, etc.

The translations are also specifications that generically describe how sentences in one language are translated to another language. The systematic use of model transformations is in the heart of model driven development (MDD) approaches, where software development's complexity is dealt in a systematic way.

DSLTrans [BLA+10] is a syntax-to-syntax model transformation language designed to define analyzable language translations. Moreover, this language imposes, by construction in its semantics, that all translations are confluent and terminating. These properties are in fact determinant for the analyzability of the translations due to the finite size of the translation's analysis/symbolic space. The analysis of a translation tries to figure-out what are all the possible general relations between syntactic structures (i.e patterns) expressed in the source language, and its translated versions expressed on the target language. These structural (rather syntactic) correspondences between languages can be automatically checked on a given translation by means of a verification method [LBA10].

However, it is still not clear how we can assert the correctness of the translations' structural correspondence set in w.r.t the underlying semantics of both source and target languages — i.e the translation might be semantically wrong if a model in the source language do not have exactly the same meaning after being translated to the target language.

In this paper, we discuss how can we assert about the correctness of a translation, by establishing a generic semantic relation between both source and target languages' semantic definitions. In Section 2 we present what are the main research trends in this subject. Then, in Section 3, we give an theoretical overview of the approach while introducing the involved basic concepts. In Section 4, we demonstrate our approach by means of an experiment that consisted in: *(i)* building a translation specification between the State Machine Language and the Petri Net Language; and *(ii)* asserting about its correctness w.r.t. their semantics expressed by means of Structured Operational Semantics (SOS). In Section 5, we discuss the results of the experiment, and we elaborate on how this approach can be used to assert the correctness about language translations in general. Finally, in Section 6 we conclude and present future insight on the research on this subject.

2 Related Work

The quality of model transformations, and in particular the quality of translations is a subject of great interest. [Kus04] presented some important guidelines and properties that need to be checked during the validation of some model transformation: syntactic correctness of both input and output models; termination and confluence (unique results and determinism); semantic

equivalence or semantics preservation; safety or liveness (to ensure preservation of structural or security properties).

The proof that a model transformation is valid for any possible model expressed in some source metamodel is in general not automatic and not even easy to master for a quality engineer.

There are many examples of work on trying to analyse language transformations at the meta level by reaching proofs from the transformation rules [Con09], [BGL05]. For instance, in [Con09] an encoding of the lambda-calculus into the pi-calculus (by three simple recursive rules) is presented, as well as the proof that some semantic properties of the lambda-calculus are preserved after the encoding into the pi-calculus. Although these languages are relatively small — even minimalist in our context — the proof that these semantic relations hold between them is still not easy perform by a quality language engineer.

In order to aid the construction of the proof of semantic preservation along a set of transformation rules [ALL10] introduced a language to anotate those rules with assertions. The idea is to then pass these annotations to a reasoning framework that will derive, at the meta level, conclusions about the overall transformation. The work presented in [ABK07] aims at validating a model transformation by using the Alloy tool. In this case, Alloy simulates the transformation by generating a model example of the source language and then analyzing the results of the transformation.

The authors of [FHLN08] present a constructive fashion to automatically generate a valid transformation (the authors refers to transformations as ontology alignment) which in principle would preserve the semantic properties of the input and output models. This generation is done by using the Similarity Flooding algorithm which is based on a calculation of a distance measurement between source and target languages.

3 Overview of the approach

In our approach, we use formal models of software languages called **linguistic metamodels** (or just metamodels throughout this paper) in order to be able to decide if a given model (or just sentence throughout this paper) is a syntactically valid expression of a given language. This decision is realized by relating both the terms and their composition on a given sentence, and the terms and their composition on the metamodel. For instance, if we look at the shapes of these sentences as being graphs (i.e made out of vertices and edges relating vertices), then we can relate the expressed sentences with the metamodel of its language by means of an instance relation (also referred in the literature as the **conformance relation**). A sentence being an instance of a metamodel means that *(i)* every concept in the sentence is also present in the metamodel of the language, and *(ii)* every relation between two concepts in the sentence is also present in the metamodel of the language relating those concepts. Additional constraints are introduced to distinguish the types of these relations (e.g containment or association relations), and their cardinalities (e.g one to one associations, one to many, etc.). These metamodels could in principle be used to generate sentences on a given language. However the complete set of sentences that conforms to a language is typically infinite. Nevertheless, this conformance relation is merely syntactic, which means that the referred metamodels corresponds to the so called **abstract syntax of a language**, since they filter the structure and shape of valid expressions of that language

by only looking to their explicit structure regardless of their implicit value/meaning.

However, this implicit meaning of each and every sentence expressed in a software language can be made explicit by means of formal mathematical descriptions such as the one proposed by Plotkin [Plo04]). Plotkin proposed that all the computation steps of a valid computer program while running in hypothetical computer system can be generically described by means of a finite set of pre/pos condition rules. These rules form what we call the **Structural Operational Semantics** (SOS) of a software language. If we take a program expressed in a given software language, we can use these SOS rules to collect all of its possible computation steps, and then build up a graph which we call the program's **transition system**. In this graph, each edge is a transition generated by the conclusion of an application of a SOS rule while symbolically executing that program. These edges relates source and target vertices, where each vertex represents the computation state (i.e values in the hypothetical machine's memory) before and after that transition occurred in the symbolic execution. A path in a given transition system, is called a symbolic execution **trace**. Note also that with these SOS rules, the implicit meaning of a finite sentence cannot simply be made explicit because there may be a possible infinite amount of possible SOS rule applications (i.e its transition system can be infinite).

Despite the fact that the language sentences' transition systems can be infinite, it is still possible, for instance, to use their SOS descriptions for checking or proving that two different sentences in a language have the same meaning or value, by establishing an semantic equivalence relation between their transition systems. Examples of these relations are the strong **bisimulation-equivalence**, or some other weaker forms such as the **simulation equivalence** [Par81].

Intuitively, two transition systems are bisimilar-equivalent if all of their possible moves (execution traces) match each other.

When we perform a translation between a source sentence and a target sentence, the first thing to consider while asserting correctness of a given translation, is that those sentences may be expressed in different languages, and that translation preserves its abstraction level. Which means that the transition systems of every sentence expressed in the source language, and of its respective translated sentence expressed in the target language, are bisimilar-equivalent. In other words, to prove that a given translation is correct, we would need to verify this equivalence between the transition systems of every possible sentence expressed in the source language, and of their counter-parts in the target language.

Since a language have an infinite amount of possible sentences, this proof would never terminate. Clearly, we have to take a closer look on how the translation is being specified, and extract from it a finite amount of relevant sentence pairs (from both source and target language) in order to check the semantic equivalence of their transition systems. These relations are illustrated in the commutative diagram in the Figure 1. In practice, there may be several ways of transversing this diagram. However, in our experiment we only show one operational method to transversing it in a tractable way.

Our hypothesis, is that, if our translation under analysis is expressed in a graph-based model transformation language, then these relevant sentence pairs can be obtained from the match and apply patterns (left hand side and right and side graphs respectively) on the rules of the model transformation itself. Following this line of reason, if we compute all the possible combinations of rule applications, and multiply by all the possible combinations of composing or merging these patterns together, then we might get a finite amount of relevant sentence pairs, hence giving the

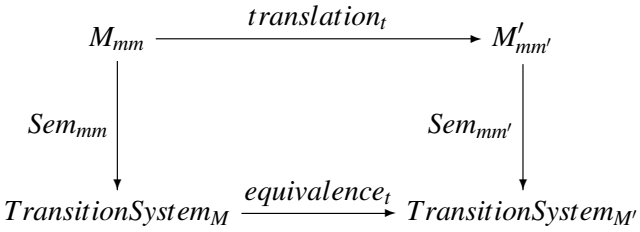


Figure 1: A commutative diagram illustrating the logical principles of the approach. M and M' are representative sentences from source and target languages of the translation. These representative sentences that can be extracted from a translation specification, where mm and mm' are metamodels identifying each language, Sem_{mm} and $Sem_{mm'}$ are each one a set of SOS rules defined for each language, and $TransitionSystem_M$ and $TransitionSystem_{M'}$ are the resulting transition system from each sentence M and M' respectively.

possibility to further check their semantic equivalence. We call these pairs of relevant sentences as symbolic states of the translation, since they represent intermediate or final relations between source and target sentences during the translation's execution.

Depending on the expressiveness of the model transformation language used to express a given translation, it might be even impossible to prove that the number of the translation's symbolic states is finite.

DSLTrans, is a graph-based model transformation language that enables the specification of translations. These translations are expressed by means of groups of rules organized in a list of sequential layers — which means that the group of rules in the first layer, is executed before every other groups. As in a regular graph-based model transformation language, these rules are formed by a left-hand-side graph (which we call the **match model** of the transformation rule), and by a right-hand-side graph (which we call the **apply model** of the transformation rule). Every translation expressed in this language is proved by construction to be confluent and terminating. Moreover, it is also proved that the number of relevant sentence pairs (translation's symbolic states) extracted from a DSLTrans' translation is finite [LBA10]. In our experiment, we will only assert the correctness of translations expressed in DSLTrans.

4 Case Study: Translating State Machines to Petri Nets

To demonstrate our approach, we present a case study of a translation between two languages: State Machines and Petri Nets.

We considered a scenario where a language engineer started to develop a domain specific language by specifying its abstract and concrete syntax, and then assigned its formal semantics by means of SOS. At some point the language engineer finds some convenience to translate the State Machine sentences into sentences expressed in the Petri Nets language. In this particular case, this translation can be very useful since it can enable the reuse of the simulation facilities given by Petri Nets modeling tools. Once the translation is specified, the final question then is

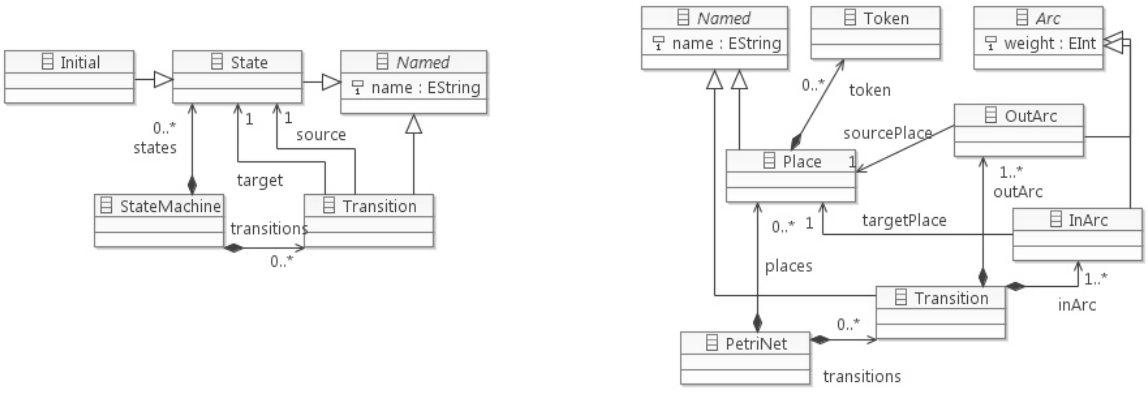
how to assert that the specified translation is correct.

One of the translation's requirements that the language engineer has to prove on its specified translation, is that all possible valid sentences in the State Machine Language have an equivalent meaning (in this case computational behaviour) with its translated petri net versions. Thus, we will use a strong equivalence relation such as bisimilar-equivalence.

4.1 Preparing the Translation

In this experiment, both the State Machine Language and the Petri Nets Language's abstract syntax were defined by means of Ecore-based metamodels [1], as shown in Figure 2.

In this experiment, the language engineer defined its translation using the DSLTrans' transformation tool, which uses EMF-based metamodel specifications in order to load state machine sentences — i.e conformant with the State Machine Metamodel— and produce petri net sentences — i.e conformant with the Petri Net Metamodel.



(i) *(ii)*

Figure 2: The State Machine Language Metamodel *(i)*, and the PetriNet Language Metamodel *(ii)*

Once the abstract syntax of the language is defined, the language engineer defines its concrete syntax and its semantics definition. The concrete syntax definition of a language usually extends the existing abstract syntax with symbols and usable metaphors that enable the domain experts to quickly understand the sentences in that language. In our examples, for readability, we prefer to use the concrete syntax versions of both state machines and petri nets sentences. As a reference, we exemplify how the same sentences look like by using or not its concrete syntax definitions in Figures 3 and 4, where we show examples of sentences expressed in the State Machines Language, and the Petri Nets Language respectively.

As mentioned before, the operational semantics of the language can be made explicit by means of a set of SOS rules. In this experiment, the language engineer defined two small semantics, one for each language. The **operational semantics of the State Machine language** is formed by the following SOS rule defined for an arbitrary State Machine sentence *s*:

[1] http://www.eclipse.org/modeling/emf/?project=emf

$$\frac{(Transition_s \xrightarrow{source} State_s) \wedge (Transition_s \xrightarrow{target} State'_s)}{cs(State_s) \xrightarrow{transition} cs(State'_s) \in TS_s},$$

This SOS rule says that if *i)* we have a *Transition* defined in a given State Machine sentence s, which is connected to both a $State_s$ (by means of a *source* relation), and to a $State'_s$ (by means of a *target* relation); and *ii)* the current state of execution happens to be $State_s$ (written $cs(State_s)$), then there will exist a *transition* in the transition system of the sentence s (written TS_s), from $cs(State_s)$ to $cs(State'_s)$. This *transition* means that it is possible to move from a current state of $State_s$ to $State'_s$ given the current state machine sentence s. Moreover we need to build the first current state ($cs(Initial_s)$) by just saying that $Initial_s \implies cs(Initial_s)$, while imposing that s needs to have exactly one *Initial* state.

The transition system of a particular sentence expressed in the State Machine Language is therefore a set of all the possible $\xrightarrow{transition}$ relations between current states (denoted as $cs(State_S)$). Also note that the transition system of the State Machine Language is finite — intuitively, given a particular state machine, the value of the current state value will range on all the defined states, and the number of $\xrightarrow{transition}$ between the current states will be bounded by the number of transitions defined in that particular state machine.

The **semantics of the Petri Net language** is defined by the following SOS rule defined for an arbitrary petri net sentence p:

$$\frac{\begin{array}{l}\forall(Transition_p \xrightarrow{outArc} OutArc') : \\ (OutArc'_p \xrightarrow{sourcePlace} Place'_p) \wedge (Place'_p, T') \in M \implies OutArc'_p.weight \leq T'\end{array}}{m(M = \{(Place_p, T)\}) \xrightarrow{transition} m(\{(Place_p, T + InArc_p.weight - OutArc_p.weight)\}) \in TS_p},$$

where $InArc_p$ is such that $(Transition_p \xrightarrow{inArc} InArc_p) \wedge (InArc_p \xrightarrow{targetPlace} Place_p)$, and $OutArc_p$ is such that $(Transition_p \xrightarrow{outArc} OutArc_p) \wedge (OutArc_p \xrightarrow{sourcePlace} Place_p)$. M is also called the

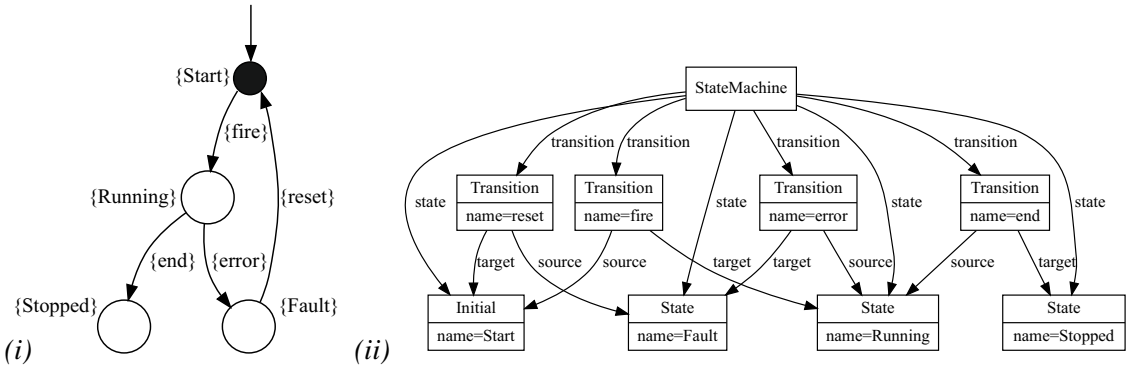


Figure 3: The standard visual representation of a state machine using state machine language's concrete syntax *(i)*, and its internal hierarchical EMF representation *(ii)*: a model instance of the petri nets metamodel.

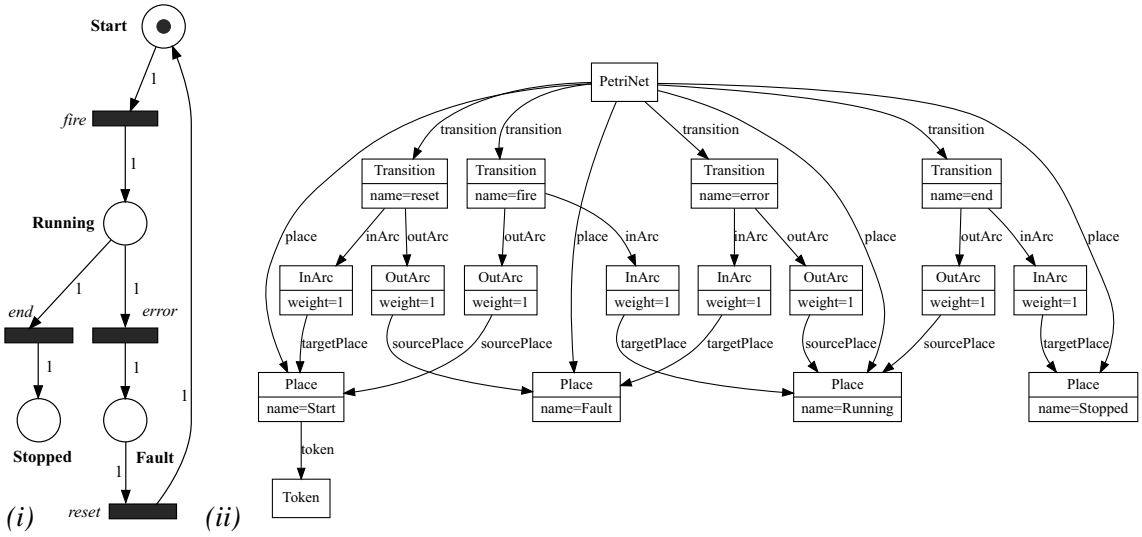


Figure 4: The standard visual representation of a petri net using the language's concrete syntax *(i)*, and its internal hierarchical EMF representation *(ii)*: a sentence instance of the petri nets metamodel. There exists very efficient verification algorithms/techniques developed to determine invariants of a petri net sentence, such as the overall number of tokens.

marking of the petri net and is initially calculated as $M = \{(Place_p, T) \mid T = |\{(Place_p \xrightarrow{token} Token_p)\}|\}$.

The semantics of a particular Petri Net expression is a set of all the possible $\xrightarrow{transition}$ relations between markings. Each marking value (denoted here as M) represents the number of tokens on each place at some point in the run-time of a particular petri net sentence. Intuitively, there may be transition systems of Petri Net sentences that may not be finite. In fact, if we consider a simple petri net sentence with a cycle that produces more tokens than it consumes, the resulting number of value combinations for its marking would be already infinite.

4.2 The Translation Specification and its Properties

Despite the fact that the two presented languages do share similar concepts, their semantics are very different in nature, as shown above. However, when the language engineer considers to specify a translation between them, there is some assumption of semantic compatibility between them. In other words, the language engineer assumes that there exists a translation where every state machine, and its translated version in petri nets, have exactly the same behaviour. Therefore everything meaningful that we can express in the State Machines language should also have the same meaning in the Petri Nets Language.

In this small experiment, the language engineer produced the translation shown in Figure 5. The patterns in both match and apply models refer to elements of the State Machine and Petri Net metamodels, respectively. The language engineer, first wrote the layer *Entities* which specifies a direct 1 to 1 mapping between all the concepts of state machines with some significant ones

from the petri nets' language (e.g *State* : *Exp* is translated to *Place*). Then, the language engineer wrote the second layer *Associations* specifying the translation between all the expressible (relevant) state machine term compositions with petri net compositions. These compositions may involve the introduction of new concepts. For instance, to translate the relation named *source* between *State* : *Exp* and *Transition*, the language engineer had to introduce the concept of *OutArc*.

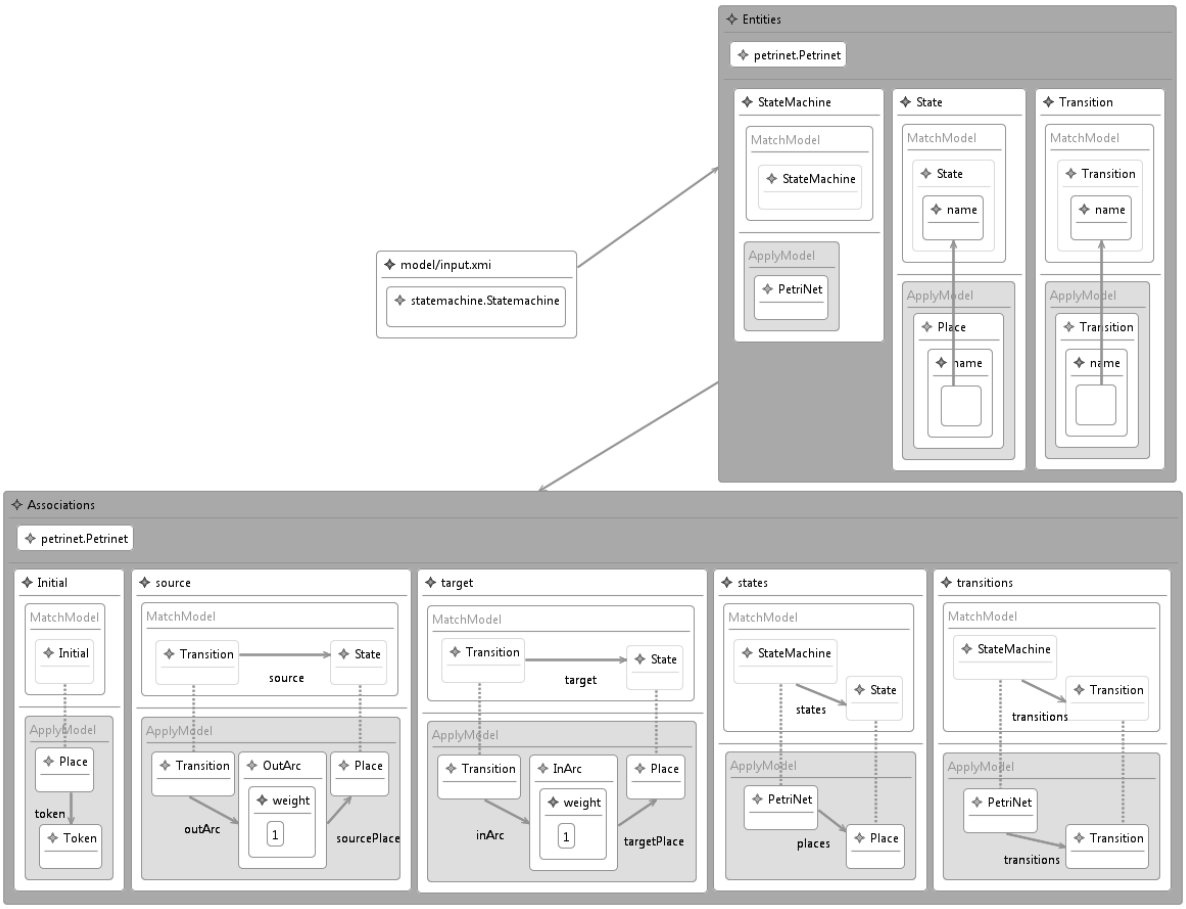
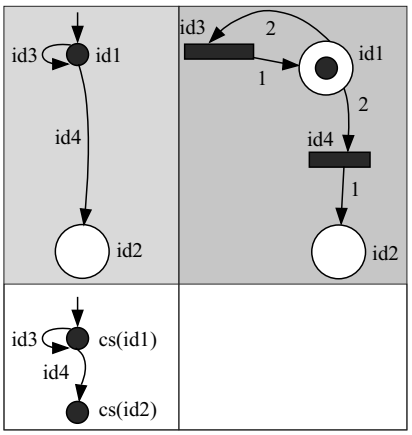


Figure 5: A DSLTrans' translation specification between the State Machine Language and the PetriNet Language.

At first glance, this translation seems to make sense, but it remains an intuition based on the empirical knowledge of the language engineer. For instance, consider that the translation has in fact a mistake, where the weight of the produced *OutArc* in rule *source* is not 1, but 2. This specification would in fact produce meaningless petri nets from meaningful state machines. If we generate a translation's symbolic state which is a pair (M, M') of relevant sentences from our translation, as shown in Figure 6 *(i)* — the resulting transition systems TS_M and $TS_{M'}$ are not bisimilar-equivalent — in fact, in this case, $TS_{M'}$ is empty, since the resulting petri net is deadlocked. A similar symbolic state resulting from the analysis of the correct translation presented

in Figure 5, is shown in Figure 6 *(ii)*.

(i)



(ii)

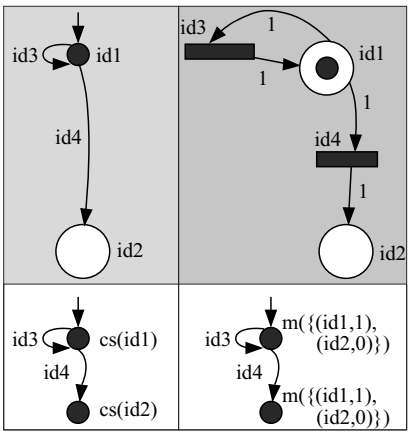


Figure 6: DSLTrans' symbolic states (top) annotated with their transition systems (bottom). On the left *(i)*, a symbolic state of an erroneous translation. On the right *(ii)*, a symbolic state of the correct one.

Both of the presented symbolic states were a result of an analysis over the presented translation specifications (both the erroneous and the correct one), where the rule *source* was combined with rule *target*, exploring the different cases where both the source nodes and target nodes of the transitions are the same or not. In the symbolic state presented in Figure 6 *(ii)* the bisimilar-equivalence relation R between both state machine and petri net's transition systems the can be expressed as the following:

$$(cs(id1)), m((id1,1),(id2,0))) \in R, \text{ and } (cs(id2)), m((id2,0),(id2,1))) \in R$$

In fact, if we look into the complete set of symbolic states generated from the translation presented in Figure 5, and keep only the symbolic states which have valid statemachine sentences (i.e having only one initial state on their match pattern parts), we can express the same bisimilar-equivalence relation R in a more comprehensive way:

$$(cs(State), m(\{(Trans(State),1)\} \cup \{(Place',0) \mid Place'! = Trans(State)\})) \in R,$$

where $Trans$ is the translation function given the translation specification under analysis. In other words, all petri net sentences generated in this particular translation will be bisimilar to their original state machine counterparts. Also, during symbolic execution, they will have globally exactly one token, and this token will lie on a place which corresponds exactly to the current state of the corresponding state machine.

5 Discussion of the Results

In this section, we reason about the applicability of the approach in the engineering of DSMLs in general.

One important limitation while using this approach is that it will only work properly, in the practice of language engineering, if we are able to check semantic equivalence relations between the transition systems of each model on each pair provided by the analysis of the given translation. It is intuitive that, depending on the kind of semantic equivalence that we are trying to prove, if the source language enables sentences can have infinite sized transition systems, we can no longer use this method to assert the correctness of that translation — note that in the experiment presented in the Section 4, the source language of State Machines — as some other DSMLs — did had a finite transition system. Notice that we do not require that the transition systems of all sentences in the target language of a translation to be finite — this is due the fact that our translations are not (by definition) bi-directional, and our assumptions rely only on that.

Besides that, having the fact that DSMLs' semantics are usually realized by means of code generators without any use of operational semantics, it is questionable the use of this technique in practice. However, the intuition is that this technique can be applied if the language engineer builds up a component model from its generated code — i.e by making the generated code conformant with a component language, and by using its associated SOS semantics.

6 Conclusions and Future Work

In this paper, we presented a technique that enables the language engineers to assert the correctness of language translations. In particular, we validated a translation between a State Machine Language and Petri Nets Language. We showed that the resulting relation between their transition systems is sound w.r.t. the language engineer's intention while specifying its translation — i.e every state machine is translated to a particular kind of petri nets that have as an invariant, the fact that all of its possible markings will have a global size of tokens equal to one. Also, we discussed about the current limitations of this technique while being applied to assert the correctness of translations of DSMLs in general.

As future work, we will apply this technique to component models built from code generator patterns, in order to study what are the main restrictions that have to be applied to the component language so that the resulting semantic domain of that component language becomes finite, and therefore analyzable.

Acknowledgements: This work was developed in the context of the following research institution: CITI fund PEst-OE/EEI/UI0527/2011 Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) - 2011-2012

Bibliography

[ABK07] K. Anastasakis, B. Bordbar, J. Küster. Analysis of Model Transformations via Alloy. In Baudry et al. (eds.), *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2007), Nashville, TN (USA)*. Pp. 47–56. Springer, Berlin/Heidelberg, October 2007.

[ALL10] M. Asztalos, L. Lengyel, T. Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *ICST 2010: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. Pp. 15–24. IEEE Computer Society, 2010.

[BGL05] *Formal Verification of Java Code Generation from UML Models*. Fujaba Days, september 2005.

[BLA+10] B. Barroca, L. Lucio, V. Amaral, V. Sousa, R. Felix. DSLTrans: A Turing Incomplete Transformation Language. In *Proc. 3rd International Conference on Software Languages Engineering - SLE 2010*. Springer-Verlag, 2010.

[Con09] A Logical Interpretation of the Lambda-Calculus into the Pi-Calculus, Preserving Spine Reduction and Types. In *CONCUR 2009 - Concurrency Theory - Lecture Notes in Computer Science*. Volume 5710, pp. 84–98. Springer Berlin / Heidelberg, 2009.

[FHLN08] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In Czarnecki et al. (eds.), *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*. Lecture Notes in Computer Science 5301, pp. 326–340. Springer, 2008.
doi:http://dx.doi.org/10.1007/978-3-540-87875-$9_2$4

[KT08] S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling*. Wiley-IEEE Computer Society Press, March 2008.

[Kus04] J. M. Kuster. Systematic Validation of Model Transformations. In *Essentials of the 3rd UML Workshop in Software Model Engineering (WiSME2004)*. 2004.

[LBA10] L. Lucio, B. Barroca, V. Amaral. A Technique for Automatic Validation of Model Transformations. In *ACM/IEEE MoDELS 2010*. Springer-Verlag, 10 2010. URL=http://http://models2010.ifi.uio.no/.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. Pp. 167–183. Springer-Verlag, London, UK, 1981.
http://dl.acm.org/citation.cfm?id=647210.720030

[Plo04] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61:17–139, 2004.

[Plu98] D. Plumpf. Termination of graph rewriting is undecidable. *Fundamenta Informaticae* 33(2):201–209, 1998.