# From SysML State Machines to Petri Nets Using ATL Transformations

Rui Pais[1,2,3], João Paulo Barros[2,3], and Luís Gomes[1,2]

[1] Universidade Nova de Lisboa, Faculty of Sciences and Technology, Portugal
ruipais@uninova.pt
[2] UNINOVA, Center of Technologies and Systems, Portugal
lugo@fct.unl.pt
[3] Instituto Politécnico de Beja, Escola de Superior Tecnologia e Gestão, Portugal
jpb@uninova.pt

**Abstract.** The ATLAS Transformation Language (ATL) is a well-known hybrid model transformation language that allows both declarative and imperative constructs to be used in the definition of model transformations. In this paper, we present ATL transformations providing an integrated structural description of the source and target metamodels and the transformation between them. More specifically, the paper presents translation rules of Systems Modeling Language (SysML) state machines models into a class of non-autonomous Petri net models using ATL. The target formalism for the translation is the class of Input-Output Place Transition Nets (IOPT), which extends the well-known low-level Petri net class of Place/Transition Petri nets with input and output signals and events dependencies. Based on this Petri net class, a set of tools have been developed and integrated on a framework for the project of embedded systems using co-design techniques. The main goal is to benefit from the model-based attitude while allowing the integration of development flows based on SysML state machines with the ones based on Petri nets.

**Keywords:** ATL, Transformation Models, SysML, UML, State Machines, Petri Nets, PNML, MDE, MDA, IOPT.

## 1 Introduction

The increasing complexity of new-generation systems raises major concerns in various critical application domains, in particular with respect to the validation and analysis of performance, timing, and dependability-related requirements. Model-driven engineering (MDE) [1] approaches aimed at mastering this complexity during the development process have emerged and are being increasingly used in industry. They address the problem of complexity by promoting reuse and partial or total automation of specific phases of the development process.

By taking advantage of Petri nets visual representation and precise semantics, this paper contributes to their use as an intermediate formalism between SysML behavior models and code generation. More specifically, it presents a tool that transforms a

L.M. Camarinha-Matos et al. (Eds.): DoCEIS 2014, IFIP AICT 423, pp. 227–236, 2014.

Systems Modeling Language (SysML) state machine, to an Input-Output Place Transition Net (IOPT) [2] target model. The process of transformation — the production of an XMI [3] file — is realized using the Atlas Transformation Language (ATL) [4]. The XMI file will be subsequently used as an entry to a Model Driven Architecture (MDA) [5, 20] process.

Present work benefits from the results of previous projects where a set of tools was created. These tools allow creation of IOPT net models, as well as many others operations. Ultimately, they also permit code generation for different platforms. Those tools rely on an Ecore metamodel for the IOPT class, already presented in [2].

Translation techniques to convert state machines elements into correspondent Input Output Place Transition Nets items were already presented elsewhere [7]. Later, these translations were extended to include other state machines elements and several strategies to translate state machines with pseudostate history attribute were also presented [8].

This research extends previous works [7, 8] with the implementation of translation techniques, using ATL, to transform SysML state machine models to IOPT net models. The integration of these translations on a Petri net-based framework for the development of embedded systems using co-design techniques permits the use of SysML state machines as an additional modeling language.

The paper is structured as follows: Section 2 presents motivation, innovations and its relation to Collective Awareness Systems, while Section 3 describes some important concepts of Model to Model transformations. The implemented transformations rules using ATL are presented in Section 4. Finally, Section 5 presents topics for discussion and Section 6 concludes.

## 2 Relation to Collective Awareness Systems

An awareness system can be defined as a system intended to help people construct and maintain awareness of each other activities, context or status, even when the participants are not co-located [9].

As testified in [10], "Internet has changed the way we develop, perform and understand business", hence it is urgent to innovate and exploit the full potential of the Future Internet. To achieve this goal, a set of recommendations is provided and an analysis of important areas to consider is presented. More specifically, the analysis of Collective Awareness Platforms for Sustainability and Social Innovation (CAPS), points out that its basic layer includes (smart) objects that capture the environment reality. Many of these (smart) objects are embedded systems, which are frequently specified using Petri Nets. These are used for modeling and analyzing complex systems that exhibit characteristics of concurrency, synchronization, simultaneous, distributed, resource sharing, etc., taking advantage of tools that permit modeling visualization, simulation, property verification (e.g. deadlock, starvation, bottlenecks, and execution time), and code generation.

With this research, we contribute to the development of embedded systems allowing system specification using a well-known modeling formalism: SysML state machines. With better specification we are promoting better embedded systems, the core of smart objects.

# 3 Model to Model Transformations

Software development is becoming more and more complex with the increasing complexity of system requirements, necessity to integrate frameworks, libraries, communication platforms, etc.

Maintenance, changing requirements, production cost, specification reusability, and a lot of other factors contribute to a demand to improve software specification. One way to reduce technical complexity is the use of the Model-Driven Software Development paradigm, as it facilitates a more abstract specification of software based on modeling languages [11].

In the context of model-driven engineering, models are the main development artifacts and model transformations are among the most important operations applied to models. Model-to-model transformations constitute an important ingredient in model-driven engineering.

The following sections present the main concepts used in this context.

## 3.1 Ecore

The model used to represent models in the Eclipse Modeling Framework (EMF) is called Ecore. Ecore is itself an EMF model, and thus is its own metamodel [12].

Ecore is a modeling language (in fact a meta-meta-modeling language) to describe domain specific meta-models. It is used to define all entities that will exist on the domain specific models of interest, define the characteristics of these entities, as well as their relationships.

## 3.2 ATL Language

The ATLAS Transformation Language is a model to model transformation language originally developed as a response to the Object Management Group (OMG) Request for Proposals for the Query/View/Transformation (QVT) standard [13], implemented in the Eclipse modeling tools.

ATL is a well-known hybrid model transformation language that allows fully declarative, hybrid, or fully imperative constructs to be used in transformation definitions.

An ATL transformation is unidirectional and operates on read-only source models to produce write only target models. This way, source model can be navigated during transformation, but it is neither possible to write on the source model, nor to navigate target model. Yet, ATL provides automatic traceability links between target and source elements.

ATL model transformation process is illustrated on Figure 1. This provides an overview of the transformation process used for the generation of an IOPT model (IOPTnet.xmi) from a SysML state machine model (StateMachine.xmi) through a transformation model (SM2IOPT.atl). In this diagram, the dotted line specifies model to model transformation; the normal arrows specify conformance between source state machine model (StateMachine.xmi) and its metamodel (StateMachine.ecore), as

well as between this metamodel and the Eclipse Modeling Framework (EMF) metametamodel. On the same way, the generated model (IOPTnet.xmi) conforms to its metamodel (IOPT.ecore), and the latter conforms to the EMF metametamodel.

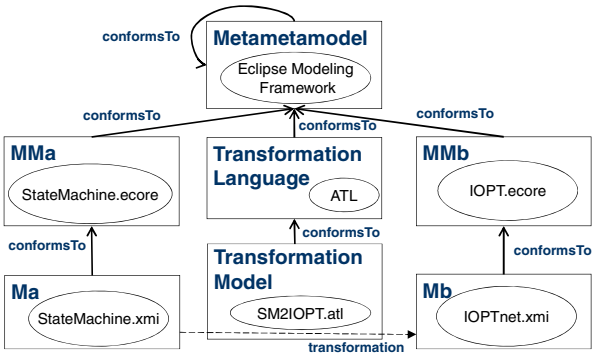


**Fig. 1.** ATL Model Transformation (adapted from [21])

### 3.3 SysML State Machines

The Systems Modeling Language (SysML) is a general-purpose modeling language that supports the specification, analysis, design, verification, and validation of a broad range of complex systems, which may include hardware, software, information, processes, personnel, and facilities [14]. The respective State Machine package defines a set of concepts that can be used for modeling discrete behavior through finite state transition systems [14].

### 3.4 Input-Output Place Transition Nets

Petri nets have been successfully used for concurrent systems specification [15] for the analysis of behavioral properties and performance evaluation, as well as for systematic construction of discrete event simulators and controllers [16].

The Input-Output Place Transition class (IOPT) extends place-transition nets with non-autonomous constructs, allowing explicit modeling of input and output events and signals [2]. The Ecore metamodel for the IOPT net class was already presented in [2, 17].

## 4 Transformations Rules Using ATL

In this section, we present a set of transformation rules and helpers that allow the generation of IOPT models from SysML state machines models. The result of each transformation is an XMI file on PNML format.

Some translation techniques to convert state machines elements into correspondent Input Output Place Transition Nets items were already presented elsewhere [7, 8].

The respective implementations are here presented. Yet, as the transformation application is extensive, we only show some code blocks, representative of interesting transformation aspects.

**Transformation rules** are the core of a model transformation. A transformation rule can be implemented with declarative, fully imperative, or hybrid constructs.

The declarative style of specifying transformations are encouraged as best practice [18], letting imperative features of the language to the most complex transformations when declarative constructs are not sufficient.

Declarative transformations rules use pattern matching to match the source model elements by type. A rule guard permits an additional filtering on pattern matching. For each matched element one or more target model elements are created.

Transformations traceability is an important language feature that offers trace links between source and target models, and between rule applications. This is important, as it allows one rule to reference elements created in other rules.

The matching rule 'package2PetriNetDoc' is one of simplest transformation rule.

```
rule package2PetriNetDoc {
    from
        inn : SysML!Package(inn.isPackage())
    using {
        ruleName:String= 'package2PetriNetDoc'.debug('!Starting);
    }
    to
        petriNetDoc :  PNML!PetriNetDoc(
            nets <- thisModule.allStateMachines->collect (p |
                            thisModule.resolveTemp (p, 'net'))
        )
    do {
            ruleName.debug('!Ending rule');
        }
}
```

For each source model element of type 'Package' it collects the traceability links generated by target pattern 'net'; it uses the imperative parts 'using' and 'do' for debug proposes. For each source model element of type 'Package', this rule creates a target 'PetriNetDoc'. This rule uses the 'allStateMachines' helper that is presented later in this paper.

To illustrate and exemplify the implemented translations, we present the translation of state machine **choice pseudo-state**.

A choice pseudo-state has one incoming and several outgoing edges, and it is used as a switch of control flow based on zero or more guard condition. Depending on the guards' conditions, the control flow is redirect from one incoming edge to exactly one outgoing edge. Figure 2 presents a simple state machine with a choice pseudo-state, and Figure 3 presents its translation to an IOPT net.

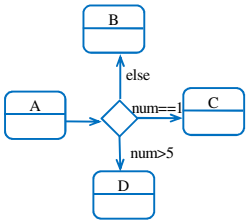


**Fig. 2.** State machine with choice pseudo-state

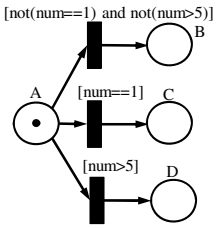


**Fig. 3.** Translated **IOPT net**

This translation was implemented by three main rules:

1. One rule that translates state machine states to IOPT places;
2. One rule to translate state machine choice guards;
3. One rule to translate state machine choice state to a set of IOPT transitions and arcs. This last transformation rule is presented next.

Matching rule 'choicePseudostate' transforms a source 'choice pseudo state' into a set of target model transitions, a set of input arcs, and a set of output arcs. This rule calls various 'unique lazy rules' to generate arcs on target models. The 'using' declarative part allows the declaration of variables that increase code legibility by giving names to different elements used in transformations.

```
--- join vertices serve to merge several transitions emanating
--- from source vertices in different orthogonal regions.
rule choicePseudostate {
    from
        inn: SysML!Pseudostate(inn.isChoicePseudostate())
    using
    {
        sourceTransition : SysML!Transition = inn.incoming.first();
        targetTransitions : Sequence(SysML!Transition) = inn.outgoing;
        sourceState : SysML!State = sourceTransition.target;
        targetStates : Sequence(SysML!State) = targetTransitions->
                                               collect(tr | tr.target);
    }
    to
        trans:distinct PNML!Transition foreach(tr in targetTransitions)(
            name <- thisModule.createTransitionName(sourceTransition,
                    sourceVertex, inn),
            priority <- thisModule.iopt_createPriority(1)
        ),
        inArc: distinct PNML!Arc foreach(tr in targetTransitions)(
            id <- thisModule.createInArcName(tr, sourceState),
            source <- thisModule.createIoptInRefPlace(sourceState),
            target <- thisModule.createIoptInRefTransition(tr)
        ),
        outArc: distinct PNML!Arc foreach(tr in targetTransitions)(
            id <- thisModule.createOutArcName(tr, tr.target),
            source <- thisModule.createIoptOutRefTransition(tr),
            target <- thisModule.createIoptOutRefPlace(sourceState)
        )
}
```

**Helpers** can be used to define attributes, constants, and methods like functions. They can be reused throughout several transformations and in different contexts as utility functions, rule guards, and model properties analysis, as well as to navigate the source model. A task that is common to many transformation examples is the definition of **helpers** to get all elements of a specific type. Next, we present some of these helpers.

(1) Helper 'allStatesMachines' gets all state machines from a source model.

```
--- helper to get all statemachines
helper def: allStateMachines: Set(SysML!StateMachine) =
       SysML!StateMachine -> allInstances();
```

(2) The matching between source models elements and transformation rules is defined by conditions guards defined on rules level. As these conditions are normally extensive, we have created **guard helpers** to increase overall legibility. Helper 'isJoinPseudostate' is a simple guard helper that verifies if vertex element is a pseudostate of the 'join pseudo state' kind. In a complete state machine, a join vertex must have at least two incoming transitions and exactly one outgoing transition [19].

```
--- Verify if pseudostate is of kind 'join pseudo state'
helper context SysML!Vertex def: isJoinPseudostate(): Boolean =
       self.oclIsTypeOf(SysML!Pseudostate)
       and self.kind = #join
       and self.incoming->size() >= 2
       and self.outgoing->size() = 1;
```

(3) Transforming elements from source model to target model, normally, doesn't depend only on the kind of source item, but also on the context where it is inserted and/or on the context of interconnected elements. It was necessary to create a vast number of **functional helpers** to navigate interconnected elements and to obtain the necessary collections or properties. Functional helper 'getOwners' gets all owners of context element, which can be of any type, and returns a sequence of all owners from context element to root item.

```
--- Get all owners of a given element
helper context OclAny def: getOwners(): Sequence(SysML!Element)=
    let itemOwner : SysML!Element =  self.owner in
       if self.oclIsUndefined() then
              Sequence {}
       else
              if(itemOwner.oclIsKindOf(SysML!Element)) then
                Sequence {self}->union(itemOwner.getOwners())
              else
                Sequence {}
              endif
       endif;
```

Previous helper ‘getOwners’ is used by various helpers to determine its relation of inclusion with other types of items. Helper ‘isInsideCompositeState()’ is one of these examples: it verifies if a transition is inside a composite state, which means that it must be inside of a composite state with or without a region inside it. This helper calls helper ‘isCompositeState()’ to determine if SysML element is of the composite state kind.

```
--- Verify if a transition is inside a composite state
helper context SysML!Transition def : isInsideCompositeState():
Boolean =
    let elements : Sequence(SysML!Element) = self.getOwners() in
        if(elements.size() >= 2) then
            elements.at(1).isCompositeState() or
            elements.at(2).isCompositeState()
        else
            elements.at(1).isCompositeState()
        endif;
```

## 5 Discussion

When implementing a translation from a source model to a target model, it is necessary to create rules that match source elements and generate target elements. When there is no clear relation between them, it is not easy to identify which elements from source should be used, and which matched element should be generated.

Currently ATL has four compilers: 2004, 2006, 2010, and EMF Transformation Virtual Machine (EMFTVM). Present research was implemented with compiler version 2010, which has important limitations on traceability (tracing mechanism), not permitting the use of all types of rules as they lack the tracing mechanism. Moving to EMFTVM, which has advanced language features, will allow some code improvements. ATL documentation is spread along Eclipse ATL site, wikis, papers, and forums. A book or some other documentation explaining "How to do transformation" and not "How to use methods/functions" is missing. Even so, using ATL as a model to model transformation language simplified all the translation process, allowing an emphasis on translation technics in opposition to programming languages issues. Preliminary results, allow us to conclude that the strategy to convert Behavior Models to Petri Nets is possible and of great interest to strengthening the assertion of Petri Nets as a formalism suitable for integration of models.

## 6 Conclusions and Future Work

The implementation of all transformations between SysML state machines and IOPT nets is a significant contribution to be added to the framework for the project of embedded systems using co-design techniques. It is now possible to go from SysML state machines to IOPT nets, verify their properties, optimize the model, generate code, visualize, and execute. As an overall view of the implementation, it is

interesting to note that 80% of all transformation rules were implemented using the declarative language style, recommended as best practice [18]. The use of variables on the imperative 'using' rules part, rule guard helpers, and called rules, allowed an overall good code readability. As near future work, it is necessary to enhance understandability of transformations and improve their correctness, which will lead to the identification and analysis of the used modeling patterns, as well as refactoring and abstraction mechanisms on model to model transformations. Finally, formal techniques for checking semantics equivalence of MDA transformations [20] can be used to validate implemented transformations.

## References


1. Gasevic, D., Djuric, D.: Devedzic, V., Selic, B.V., Bézivin, J.: Model Driven Engineering and Ontology Development. Springer (2009) ISBN-13: 978-3642101342
2. Moutinho, F., Gomes, L., Ramalho, F., Figueiredo, J., Barros, J.P., Barbosa, P., Pais, R., Costa, A.: Ecore Representation for Extending PNML for Input-Output Place-Transition Nets. In: IECON 2010 - 36th Annual Conference of the IEEE Industrial Electronics Society, Phoenix, AZ, USA, pp. 2010–2036 (2010)
3. OMG: OMG MOF 2 XMI Mapping Specification. v2.4.1. (2013), http://www.omg.org/spec/XMI/
4. ATL - A Model Transformation Technology, http://projects.eclipse.org/projects/modeling.mmt.atl (accessed on December 30, 2013)
5. OMG: MDA - The Architecture of Choice For A Changing World (2013) , http://www.omg.org/mda/
6. Gomes, L., Barros, J.P., Costa, A., Nunes, R.: The Input-Output Place-Transition Petri Net Class and Associated Tools. In: INDIN 2007 - 5th IEEE International Conference on Industrial Informatics, Vienna, Austria, Julho 23-26 (2007)
7. Pais, R., Gomes, L., Barros, J.P.: Towards Statecharts to Input-Output Place Transition Nets Transformations. In: Camarinha-Matos, L.M. (ed.) Technological Innovation for Sustainability. IFIP AICT, vol. 349, pp. 227–236. Springer, Heidelberg (2011)
8. Pais, R., Gomes, L., Barros, J.P.: From UML State Machines to Petri nets – History Attribute Translation Strategies. In: 37th Annual Conference on IECON 2011. IEEE Computer Society (2011)
9. Markopoulos, P., Mackay, W.: Awareness Systems - Advances in Theory, Methodology, and Design. Springer (2009) ISBN 978-1-84882-476-8
10. Future Internet Enterprise Systems (FInES): Embarking on New Orientations Towards Horizon 2020 (2013)
11. Lochmann, H.: HybridMDSD: Multi Domain Engineering with Model Driven Software Development Using Ontological Foundations, PhD Dissertation (2010)
12. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2008) ISBN-13: 978-0-321-33188-5
13. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1 (2011)
14. OMG: OMG Systems Modeling Language, v 1.3 (2012), http://www.omg.org/spec/SysML/

15. Choppy, C., Petrucci, L., Reggio, G.: A Modelling Approach with Coloured Petri Nets. In: Kordon, F., Vardanega, T. (eds.) Ada-Europe 2008. LNCS, vol. 5026, pp. 73–86. Springer, Heidelberg (2008)
16. Zurawski, R., Zhou, M.C.: Petri Nets and Industrial Applications: A Tutorial. IEEE Transactions on Industrial Electronics 41(6) (1994)
17. Ribeiro, J., Moutinho, F., Pereira, F., Barros, J.P., Gomes, L.: An Ecorebased Petri Net Type Definition for PNML IOPT Models. In: INDIN 2011 - 9th IEEE International Conference on Industrial Informatics, Caparica, Lisbon, Portugal, pp. 777–782 (2011), doi:10.1109/INDIN.2011.6034992 ISBN 978-1-4577-0434-5
18. Allilaire, F., Bézivin, J., Jouault, F., Kurtev, I.: ATL - Eclipse Support for Model Transformation. In: Proc. of the Eclipse Technology Exchange Eorkshop (ETX) at ECOOP (2006)
19. OMG: Unified Modeling Language™ (OMG UML), Superstructure. v2.4.1, `http://www.omg.org/spec/UML/`
20. Barbosa, P., Ramalho, F., Figueiredo, J., Junior, A., Costa, A., Gomes, L.: Checking Semantics Equivalence of MDA Transformations in Concurrent Systems. Journal of Universal Computer Science 15(11), 2196–2224 (2009), doi:10.3217/jucs-015-11-2196
21. Jiang, M., Ding, Z.: From Textual Use Cases to Message Sequence Charts. In: Information Engineering and Applications. Lecture Notes in Electrical Engineering, vol. 154, pp. 732–739. Springer (2012)