An MDE Approach for Reasoning About UML State Machines Based on Constraint Logic Programming

Beatriz Pérez
Department of Mathematics and Computer Science,
University of La Rioja,
Logroño, Spain.
Email: beatriz.perez@unirioja.es

***Abstract*—Model Driven Engineering promotes models as primary artifacts in the software engineering development process. Such models must conform to a metamodel and held associated constraints which restrict their validity. The verification of models against such requirements becomes therefore a fundamental activity to ensure the quality of a system. In this context, the Unified Modeling Language (UML) constitutes one of the most commonly used modeling languages to represent both static and dynamic aspects of software systems. Nevertheless, while the formalization and analysis of static models has motivated a significant number of proposals, it far exceeds the research done on dynamic models, specially on UML state machines, considered to be the mainstay to represent the dynamics of a system. We have defined a proposal to reason about UML state machines based on Constraint Logic programming (CLP), using *Formula* as model finding and design space exploration tool. We show how to translate a UML state machine model into a CLP program following a Meta–Object Facility (MOF) like framework. Furthermore, we enhance our proposal by giving support for the automatic translation of state machines to Formula specifications, based on a Model Driven Engineering (MDE) approach. The proposed framework can be used to reason and validate UML state machine designs by generating valid sets of execution state configurations and checking correctness properties, using Formula as model exploration tool.**

Keywords*—*UML state machines, OCL, Constraint Logic Programming, reasoning, MDE

I. Introduction

Model-Driven Engineering (MDE) has been promoted for some time as a solution to handle the increased complexity of software development. In the MDE paradigm, models constitute the cornerstone components during the software development process. Such models must conform to a metamodel and held associated constraints which restrict their validity. Effective verification of models against such requirements becomes therefore a fundamental activity to ensure the quality of a system. In the context of MDE, the Unified Modeling Language (UML) [1] has been widely accepted as the de-facto standard object-oriented software modeling language. In particular, UML is widely used in software design to specify both the static and dynamic aspects of object oriented systems, where UML Class Diagrams and UML State Machines are considered to be the mainstay to represent the statics and dynamics of a system, respectively.

As any other software artifact, software models may contain design flaws. Unfortunately, in some occasions such possible design defects are not detected until the later implementation stages, thus increasing the cost of development [2], [3]. This situation requires a wide adoption of formal methods as well as of verification and validation approaches. In this line, there have been remarkable efforts to formalize UML semantics, in order to address and solve the ambiguity, uncertainty and underspecification issues detected in UML semantics. Nevertheless, while the formalization and analysis of static models has motivated a significant number of proposals [2], [4], [5], [6], [7], [8], [9], [10], it far exceeds the research done on dynamic models, specially on UML state machines or on any other variant of Harel statecharts [11], [12]. In many of such proposals, the formalization and analysis of UML artifacts is accomplished carrying out a translation to another language that preserves the semantics. The resulted translation can be used to reason about the original model by checking predefined correctness properties about the original model [3].

In this paper, we extend the work we presented with I. Porres in [10], [13], which proposes an overall framework to reason about UML Class diagrams annotated with OCL, to give also support to UML State Machines. In particular, in this paper we propose a framework to reason about UML State Machine models based on the *Constraint Logic programming (CLP)* paradigm. As in [10], [13], we use Formula [14] as model finding and design space exploration tool, which is based on algebraic data types and CLP. More specifically, we show how to translate a UML state machine model into a CLP program following a Meta–Object Facility (MOF) like framework. Once a UML state machine model is translated into the Formula language, the Formula tool can be used, for example, to prove the *reachability* of specific states of the state machine or to check for *consistency requirements* of the state machine definition. Furthermore, in order to provide full support for the automatic translation of state machines into Formula, we have included an additional menu option in the Eclipse plugin we presented in [10], to easily and automatically carry out such translation. Our framework can be used to reason and validate UML state machine designs by generating valid sets of execution state configurations and checking correctness properties, using the model exploration tool Formula. We illustrate the usefulness and effectiveness of our approach by applying it to a particular case study.

The paper is structured as follows. Section II provides a brief introduction to UML State Machines and presents a simple case study we use throughout the paper. Section III gives an overview of our proposal for the translation of UML

 ISBN: 978-1-61208-367-4

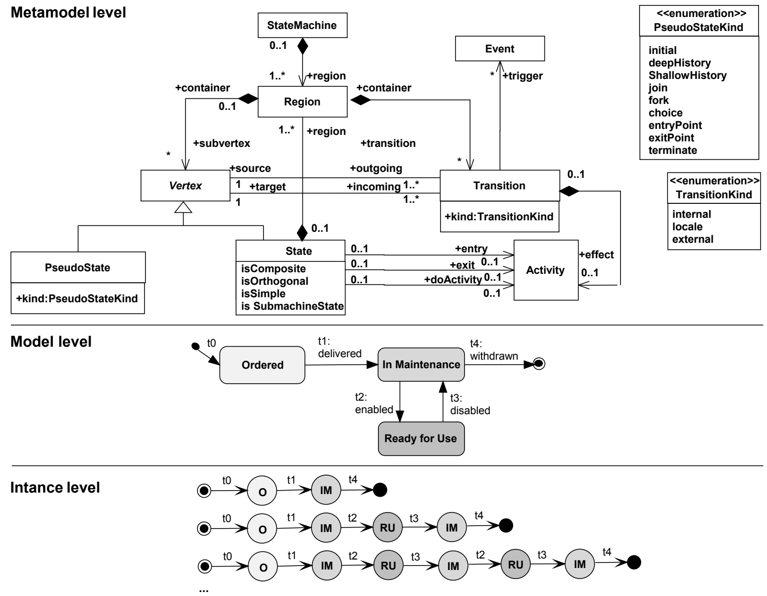

Figure 1: MOF model levels concerning UML State Machines applied to our case study.

state machines to Formula. Section IV explains the application of our proposal, and illustrates the usefulness of our approach by applying it to our case study. Related work is discussed in Section V. Finally, Section VI contains our main conclusions.

II. Backgrounding and Case Study

In this section, we present general background information of UML State Machines, together with the case study we use throughout the paper. In particular, we illustrate UML State Machines with the help of Fig. 1 in which we represent three of the MOF model levels concerning UML State Machines, applied to our case study: the *Metamodel level*, the *Model level*, and the *Instance level*. In particular, we show an excerpt of the UML State Machine metamodel (see the top side of Fig. 1), and the specific state machine model of our case study (see the center side of Fig. 1). This state machine model has been extracted from [15], which we have slightly modified to cover basic aspects of UML state machines for explanation purposes. In particular, this state machine represents the basic states that an object airplane can be in during the course of its life.

As we show in the excerpt of the UML State Machine metamodel depicted on the top side of Fig. 1, a state machine consists essentially of *states*, *transitions* and various other types of vertexes named *pseudostates* [1]. Firstly, *states* denote a situation of objects during which some condition holds. There are three kinds of states: *simple*, *composite* or *submachine*. *Simple* states are characterized by not having substates, while *composite* states are divided into *orthogonal composite states*, to model concurrent behaviors where several states are active simultaneously, and *simple composite states*, to specify that only one of their substates must be active. *Submachine* states are used basically as a way to encapsulate states and transitions so that they can be reused. In our case study, we represent that, over the course of the life of an object plane, it can take up three simple states: *Ordered*, *In Maintenance*, and *Ready for Use*. The valid set of states that the object can be active in, at a specific moment in time during the execution of the state machine, is known as *state configuration*.

On the other hand, a *transition* is the mechanism by which an object leaves a state configuration and changes to a new state configuration. A transition can be triggered by some *event*. In our case study, if the event *deliver* occurs, and the plane is the state *Ordered*, it changes to the state *In Maintenance*, nothing happens if the plane is in any other state than *Ordered*. Particularly, a *transition* is a directed relationship between a source vertex and a target vertex, where these vertexes can be either pseudostates or states. A *pseudostate* is an abstraction used to connect multiple transitions into more complex state transitions paths. There are several kinds of pseudostates (such as initial, join and fork pseudostates). An example of an initial pseudostate is shown in our case study of Fig. 1 depicted by a filled circle, representing the creation of the object plane. Additionally, *composite* states can have one or more *regions* which are considered as simple containers of a connected set of substates, pseudostates and transitions.

Finally, the sequence of state configurations an object can go through during its lifetime is known as *execution trace*. For example, on the bottom side of Fig. 1 we show three of the *execution traces* a plane can be during its lifetime. For a more complete explanation of state machines, we refer to [1].

III. UML State Machines to Formula Translation

Our proposal for reasoning about UML State Machines has some similarities with the approach we presented in [10], [13] for reasoning about UML Class Diagrams, but there is a subtle and essential difference between them. In both proposals we represent the corresponding UML metamodel and model (related to Class Diagrams and State Machines, respectively) in the Formula language, resulting a translation that can be used for several purposes. For example, the resulted Formula specification can be used to rigorously reason about the model's design, by checking predefined correctness properties about the original model such as the lack of redundant constrains. Additionally, the Formula specification can be used to inspect the model, in order to search for conforming object models and to choose those which better fit the domain needs. Nevertheless, while in [10], [13] we aimed at finding sets of classes' instances conforming the class diagram, in the case

 ISBN: 978-1-61208-367-4

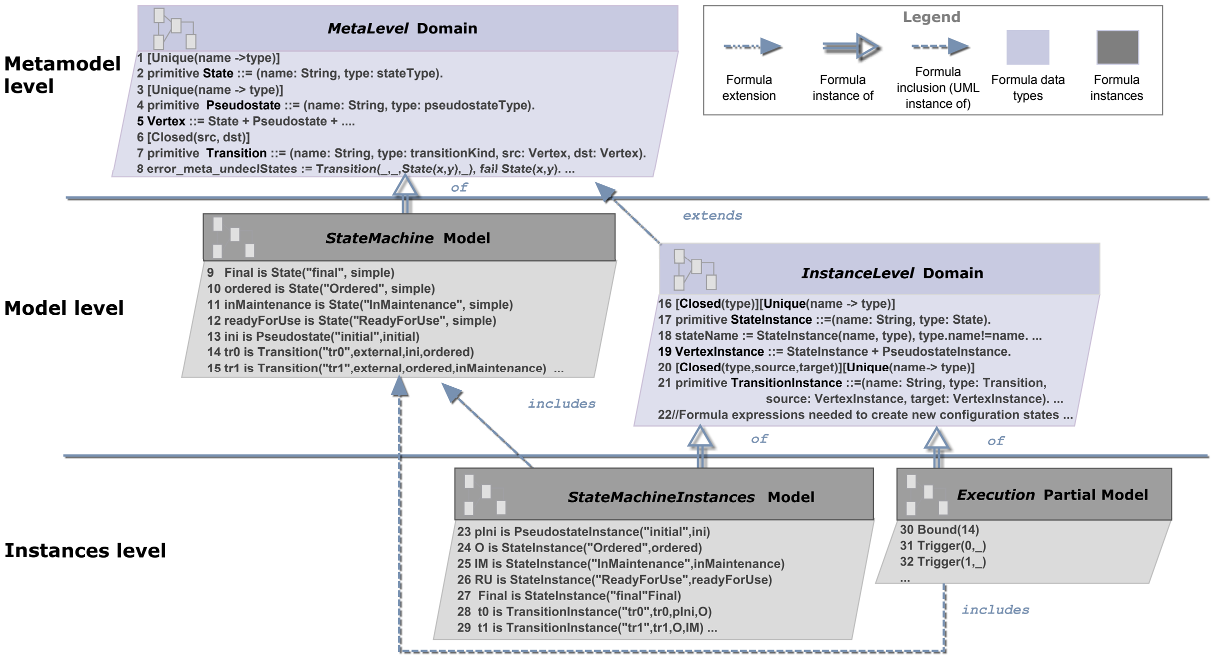

Figure 2: Formula specifications defined for the representation of our case study in Formula.

of UML State Machines we are interested in reasoning and validating UML State Machine designs by generating possible sets of state configurations, simulating valid execution traces.

As we propose in [10], [13], our approach for reasoning about UML State Machines follows a MOF-like metamodeling approach. More specifically, our proposal defines five different Formula units which are distributed along the MOF Metamodel, Model and Instance levels [1]. In order to have a better understanding of our proposal, in Fig. 2 we illustrate the defined Formula units, which are represented by means of rectangles. Furthermore, associated to each Formula unit, we have included, depicted by means of rhomboids, part of the specific Formula expressions that would be defined for representing the state machine of our case study. Next, we briefly explain our proposal for the representation in Formula of a specific UML state machine leaning on this figure.

A. Formula Data Types and Queries

Formula allows to represent a system by using three different units: *domains*, *models* and *partial models*. Firstly, a *problem domain FD* can be specified to formalize an abstraction of the problem that can be used by Formula to reason about the design. This type of units allows to specify abstract data types and a logic program describing properties of the abstraction [14]. For this reason, we have decided to represent the UML State Machine's constructs by means of domains (see *MetaLevel* and *InstanceLevel* domains in Fig. 2).

In particular, a Formula *domain* consists of *abstract data types*, *rules* and *queries*. Firstly, *abstract data types* constitute the key syntactic elements of Formula. They are defined by using the operator *::=*, indicating on the right hand side their properties by means of *fields*. Data types can be labeled in their definition with the `primitive` keyword, defining *primitive constructors*. As an example, in line 7 of Fig. 2 we define the `Transition` data type, which represents the `Transition` element of the UML State Machine metamodel. In particular, it defines several *fields* together with their types (such as the fields `src` and `dst` of type `Vertex`, representing the source and target vertexes of a transition, respectively). If the `primitive` keyword is omitted, the data type definition results in a *derived constructor* (see the definition of the type `Vertex` in line 5, representing the `Vertex` element of the UML metamodel). Data types are used as building blocks for defining Formula expressions (*terms* and *predicates*). Terms are the basis for defining *predicates*, which constitute the basic units of data, used for defining *queries* and *rules*. As an example of the definition of a *term*, in line 8 of Fig. 2 we show the term `Transition(_,_,State(x,y),_)`, representing all instances of the `Transition` term, where the third field is set to a fixed property (`State(x,y)`). The other fields are filled with a do not-care symbol ('_'), so that Formula will find valid assignments. In this way, this term represents any transition whose source state is a specific state (`State(x,y)`).

Based on the defined data types, *rules* and *queries* are specified as logic program expressions, ensuring the remaining constraints [14]. In particular, a *rule* behaves like a universally quantified implication, that is, whenever the relations on the right hand side of a rule hold for some substitution of the variables, then the left hand side holds for that same substitution [12]. The main aim of rules is *production*; they create new entries in the fact-base of Formula, populating previous defined types with facts representing the members in the collection presented in the rule. Rules are specified by means of the operator `:-`, indicating on the left–hand of the expression a simple term and, on the right–hand, the list of *predicates* specifying the rule (an example of a rule is shown later in this section). On the other hand, a *query*, which is constructed by means of the operator `:=`, corresponds to a rule where left–hand side is a nullary construction [12]. A *query* behaves like a propositional variable that is true if and only if the right-hand side of the definition is true for some substitution [12]. In particular, Formula defines in every domain the `conforms` standard query, where all constraints come together and define how a valid instance of the domain have to look like. Based on the *existential quantification* semantics of queries, we can use them to prove the existence of specific facts in the model. Additionally, the *universal quantification* can be achieved by verifying the negation of a query representing the opposite of the original predicate. For example, to ensure that `Transitions` are not created as connections of undeclared `States`, we firstly need to define a query representing the existence of transitions verifying the opposite (see line 8 of Fig. 2). With this query we are considering such incoherent situation as a valid state. Thus, to verify that such situation is not valid, we need to include the negation ('!') of the query in the `conforms` query of the specific domain.

Taking this into account, the defined domains contain (1) specific data types, which allow us to represent facts and generate reasoning instances of such types, and (2) queries, which restrict the valid system states by specifying forbidden states. Due to space reasons in this paper we mainly focus on explaining the defined data types. In particular, firstly we have created the *MetaLevel* domain, which mainly defines a primitive Formula data type for each meta model element *State*, *Pseudostate*, and *Transition*, together with specific Formula queries representing UML State Machine metamodel constraints (see lines 1 to 7 in Fig. 2). The definition of these types allows the tool to create Formula instances representing specific UML States, Transitions and Pseudostates at the Model level (such as the specific state *Ordered*). We note that, since the representation of the Metamodel level is the same whatever state machine is considered, this Formula domain is defined once and used for each state machine. On the other hand, to be able to represent the information generated during the execution of a state machine (that is, the state configurations which constitute the execution traces, together with the representation of the triggered transitions), we have defined specific types included in a Formula domain *InstanceLevel* (see Fig. 2), which defines types such as `StateInstance` or `TransitionInstance` (see lines 16-17 and 20-21).

B. Formula Data Types' Instances

Having defined the Formula domains with the abstractions of the problem, Formula gives the possibility of creating a *model FM* as a finite set of data type instances built from constructors defined in the associated domain *FD*, and which satisfies all the *FD* constraints [12]. In our particular case, we have defined two different Formula models. Firstly, we have created the *StateMachine* model, which contains the instances of the data types created in the *MetaLevel* domain, and which represent the specific elements of a particular state machine (see Fig. 2). For example, in line 10 of Fig. 2 we show the definition of the element `ordered`, which corresponds to a Formula instance of the constructor `State` defined at the Metamodel level. Secondly, we define the *StateMachine-Instances* model, which contains the instances of the data types defined in the *InstanceLevel* domain. In particular, such instances refer to the state and transition instances that Formula would use as constructors of the execution traces of the specific state machine. For example, in this Formula model we define instances such as *O* (see line 24), which would represent the fact that a specific airplane object has been in the state "Ordered". On the left hand of Fig. 3 we also show graphically the overall instances we would define for the case study. Taking this into account, the *StateMachine* model conforms with the *MetaLevel* domain, while the *StateMachineInstances* model conforms with the *InstanceLevel* domain.

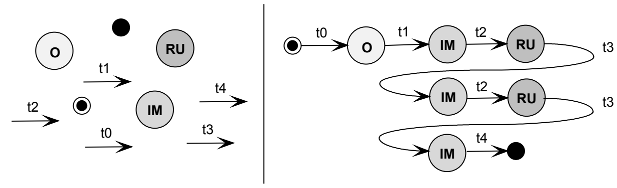

Figure 3: Instance elements and complete execution trace.

C. Logic Instructions to Simulate State Machines' Execution

Up to now, we have established the bases to be able to represent in Formula the UML State Machine metamodel, specific state machines conforming with such metamodel, as well as the concrete instances produced during the execution of a state machine and which would constitute the state machine execution traces. Nevertheless, the defined Formula data types, instances and queries are not enough to allow Formula to reason about the state machine execution, that is, to take such concrete elements and organize them into valid execution state configurations. More specifically, in addition to such instances (see the left hand of Fig. 3) and queries, we provide Formula with specific data types and rules to instruct the tool in the way to reason about such data, so that it is able to generate valid execution traces (such as the one shown on the right hand of Fig. 3). For this reason, we have completed our proposal by defining other two Formula specification blocks.

Firstly, we need to indicate Formula the way in which it has to generate a valid chain of active state configurations' instances which will constitute the valid execution traces. For this task, we have included in the *InstanceLevel* domain the definition of a new data type called `Trigger` (see lines from 3 to 5 in Fig. 4), in order to simulate the triggered of a transition. For this reason, its definition includes a field `t`, referring the moment in which the triggered takes place, and the associated `TransitionInstance` instance (see line 4). We have included the Formula constraint `[Closed(tr)]` to instruct Formula to apply a closed check to instances of the `TransitionInstance` data type, that is, using only the instances of such a type already created in the *StateMachineInstances* model. Based on the `Trigger` type, we define the type `stateConfiguration` to represent a state configuration (see line 7), and which has three fields: (1) `t`, which keeps track in time of the sequence of state configurations, (2) `v`, which refers to the specific active vertex, and (3) `traT`, which refers to the specific transition (`TransitionInstance` instance) which has been triggered to change to that state.

Additionally, in order to construct the chain of state configurations as the transitions are triggered, we have defined a Formula rule (see line 9 in Fig. 4), in order to create new entries of type `stateConfiguration` in the fact-base of Formula. As we have described previously, whenever the relations on the right hand side of a rule hold for some substitution of the variables, then the left hand side holds for that same substitution, and Formula generates the new entry corresponding to the left hand side. Taking this into account, given the current state configuration `stateConfiguration(t,src,traT)` and

```
1 domain InstanceLevel extends MetaLevel {
2  ....
3  [Unique(t ->tr)][Closed(tr)]
4  primitive Trigger ::=(t: Natural, tr: TransitionInstance).
5  triggerNumber := tri1 is Trigger(t1,tr1), tri2 is Trigger(t2,tr2), t1!=t2,tr1=tr2.
6  primitive Bound::=(t: Natural).
7  stateConfiguration ::= (t: Natural, v: VertexInstance, traT: String).
8  stateConfiguration (0,ini, traT):- ini is PseudostateInstance ("initial",
                                     Pseudostate("initial",initial)), traT="__".
9  stateConfiguration (tnext, dst,traTnext) :-
                         stateConfiguration (t, src,traT),
                         Trigger(t, TransitionInstance (tn,_,source,target)),
                         source.name=src.name,
                         target= dst, tnext=t+1,
                         Bound(end), t<end.
10 ...
11}
```

Figure 4: Generation of new state configurations.

the triggered of a transition `Trigger(t, TransitionInstance(tn,_,source,target))` whose source vertex corresponds to the current one (`source.name=src.name`), Formula creates a new fact `stateConfiguration (tnext,dst,traTnext)`, which corresponds to the new state configuration where the new state `dst` is the target vertex of the triggered transition (`target=dst`). The value of the time parameter `tnext` is also incremented by 1 for the following state configuration. Another rule is created (see line 8 in Fig. 4) to get the initial state configuration fact. We also define the `Bound` type to limit the number of transitions triggered during the state machine execution (see `Bound(end), t < end` in line 9).

Secondly, we need to instruct Formula to find valid assignments for the `TransitionInstance` appearances in the `Trigger` elements of the rule in charge of creating new `stateConfiguration` facts (see line 9 in Fig. 4). For these types of tasks, Formula defines another type of Formula units, called *partial models FPM*, in which specify individual concrete instances of the design-space or unknown parts thereof, these latter corresponding to the parts of the model *FM* that must be solved by the Formula tool [14]. For this reason, we have defined a partial model called *Execution* (see Fig. 2), in which we include as many `Trigger` terms as necessary, and which define a do not-care symbol ('_') in the field which corresponds to the `TransitionInstance` instance, so that Formula will find valid transition assignments.

IV. Application and Tool Support

In this section, we briefly explain how to use our framework in practice and apply it to our case study to illustrate its usefulness. Finally, we give some remarks of our plug-in.

The first step to apply our proposal is the translation of the specific state machine we want to reason about into the input specification language of the Formula tool. Such step is carried out by following the guidelines explained in the previous section. Having translated the UML state machine into the Formula language, the Formula finder can be used for different reasoning purposes, such as to prove the *reachability* of states or check the existence of *consistency requirements* in the state machines' definition. In particular, such requirements are represented by means of the definition of new Formula `queries`. Additionally, since the requirements are defined over the execution traces, such queries are included in the `conforms` query of the *InstanceLevel* domain, for their verification. Finally, if the system holds such requirements, the tool returns a state machine execution trace verifying all the established constraints. Otherwise, Formula will have proven that the model is unsatisfiable, that is, not execution trace is possible since some of the constraints become violated. In this latter case, the inconsistencies detected could be taken into account, for example, for the redefinition of the state machine.

In the particular case of using our proposal to prove the *reachability* of states, we can check whether there exists a path which leads to a specific state configuration. A specific use in this line is to find out whether the state machine has a valid execution trace in which the object reaches a final status (that is, there is at least a execution path in which a final status is reached, which corresponds to a *possible existence* property). As an example of application, we can test whether the final state in the state machine of our case study (represented as `stateConfiguration(_,sFinal,_)`) can be reached at some point. In this case, the following query is defined, which is included in the `conforms` query: `q1:= count(stateConfiguration(_,sFinal))=1`. Formula takes as input the state machine specification including this query, and outputs a chain of state configurations proving the reachability of the final state. In particular, Formula returns the following facts, which particularly correspond to the first execution trace depicted in the Instance level of Fig. 1:

```
stateConfiguration(0,pIni,``_'')
stateConfiguration(1,O,``t0'')
stateConfiguration(2,IM,``t1'')
stateConfiguration(3,Final,``t4'')
```

On the other hand, we can also check *consistency requirements* of state machines' definition. More specifically, we refer to consistence from a structural perspective, referring to properties that the model is expected to satisfy irrespective of its semantic content. In particular, we can verify whether the state machine exhibits a number of desired properties, obtaining at the same time the corresponding execution traces proving that the state machine holds such properties. For example, we can check whether an air plane can be available during its life time a specific number of times, obtaining the corresponding trace of state configurations. In particular, this property is checked by defining the query: `q2:= count(stateConfiguration(_,sRFU,_))=`*number*.

As for as tooling support, we have taken our *CD2Formula* plug-in presented in [10] to automatically translate specific class diagrams into Formula and we have modified it giving support for UML state machines. Finally, we have included both functionalities in an only plug-in called *UML2Formula*. Again, we have used MOFScript tool [16] which provides support for customizable model–to–text transformations. We use the UML 2.0 metamodel and the specific state machine as the model which can be defined using any UML 2.0 compliant tool that can create models in the XMI format supported by EMF (for example, the UML2 Eclipse plug-in [17] or a UML2 compliant graphical tool). As far as the Formula units generation is concerned, we have defined an only set of MOFScript transformation scripts that generates the different Formula units as stated in our proposal. The defined MOFScript transformations have been integrated into the plug–in, allowing the automatic generation of the Formula specification by means of a menu option the plug–in provides. Applying this menu option to a specific state machine, the plug–in returns a `.4ml` extension file. Later, the specific query properties to be checked have to be manually included in this file, which Formula will use for reasoning about the model.

V. Discussion and Related Work

In the past decade, there are several works which have used Constraint Logic Programming to formalize UML semantics, being limited those which tackle UML State Machines or on any other variant of Harel statecharts [11], [12], [18], [19]. In particular, there have been some proposals which aim at formalizing UML State machines which have followed a MOF–like approach to a greater or lesser extent. More specifically, authors in [11] focus on Hierarchical Finite State Machines (HFSM), which are a simplified version of UML State Machines, which consider more structural elements (such as concurrent states and pseudostates). The difference between both proposals, besides the different types of modeling languages, lies in the main goal. In particular, authors in [11] give an approach to complete partially specified dynamic models. More specifically, starting from a partial model constituted by unlinked states and transitions, they are able to find a complete state model defined from that partial model and which conforms with the HFSM metamodel. In contrast, our proposal aims at reasoning about specific state machines, not arbitrary ones, that is the reason because it starts from a complete specific state machine model instead of a partial one. In [12], authors present a metamodeling framework based on Formula and reason about *typed graphs*. In particular, they give a metamodel-based approach for representing only the `MetaNode` and the `MetaEdge` elements, at the Metamodel level, and graph `nodes` and `edges`, at the Model level, and finally reason about models. In particular, they apply their proposal to the particular case of state diagrams (where states are nodes and transitions are edges) in order to construct, similarly to the proposal in [11], well–formed state diagrams. In [19] where the author uses Alloy, a textual modelling language based on first-order relational logic, used in other works for analyzing UML class diagrams [18], gives a proposal to simulate states by specifying the notion of state on the model level, in an Alloy model, while the transition between states is given by the invocation to a UML operation.

VI. Conclusion and Future Work

In this paper, we present a framework to reason about UML state machine models based on the CLP paradigm. The main contribution of our work is the translation of a UML state machine into a Constraint Satisfaction Problem following a MOF–like framework. We enhance our approach by providing an MDE-based implementation of our translation proposal, based on our *UML2Formula* plug–in. Particularly, starting from a UML state machine representing the dynamic structure of a software system, our plug–in carries out the automatic generation of the Formula specification corresponding to such UML model, by simply choosing a menu option the plug–in provides. The proposed framework can be used to reason and validate UML state machine designs by generating valid sets of execution state configurations and checking correctness properties, using the model exploration tool Formula.

Our proposal considers basic UML State Machine elements, but the support for other commonly used elements (such as guards or composite states) constitutes a remaining work.

Acknowledgments

This work has been partially supported by the University of La Rioja (project PROFAI13/13).

References

[1] OMG, “UML 2.4.1 Superstructure Specification,” document formal/2011-08-06, August, 2012. Available at: http://www.omg.org/. Last visited on August 2014.

[2] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini, “A Formal Framework for Reasoning on UML Class Diagrams,” in *Proc. of the 13th International Symposium on Foundations of Intelligent Systems (ISMIS’02)*, ser. LNCS, vol. 2366. Springer, June 2002, pp. 503–513, ISBN:3-540-43785-1.

[3] J. Cabot, R. Clarisó, and D. Riera, “Verification of UML/OCL Class Diagrams using Constraint Programming,” in *Proc. of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW’08)*. IEEE Computer Society, April 2008, pp. 73–80, ISBN: 978-0-7695-3388-9.

[4] A. Queralt, A. Artale, D. Calvanese, and E. Teniente, “OCL-Lite: Finite reasoning on UML/OCL conceptual schemas,” *Data Knowl. Eng.*, vol. 73, pp. 1–22, 2012.

[5] M. Balaban, A. Maraee, and A. Sturm, “Reasoning with UML Class Diagrams: Relevance, Problems, and Solutions – a Survey,” March 2007, available online at: http://www.cs.bgu.ac.il/ mira/CDReasoning-07.pdf. Last visited on August 2014.

[6] M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe, “Considerations and Rationale for a UML System Model,” in *UML 2 Semantics and Applications*, K. Lano, Ed., 2009, pp. 43–60.

[7] J. Osis and U. Donins, “Formalization of the UML Class Diagrams,” in *Evaluation of Novel Approaches to Software Engineering*, ser. Communications in Computer and Information Science. Springer, 2010, vol. 69, pp. 180–192, ISBN: 978-3-642-14818-7.

[8] D. Berardi, A. Cali, D. Calvanese, and G. Di Giacomo, “Reasoning on uml class diagrams,” *Artificial Intelligence*, vol. 168, pp. 70–118, 2005.

[9] M. Gogolla, J. Bohling, and M. Richters, “Validating uml and ocl models in use by automatic snapshot generation,” *Software and System Modeling*, vol. 4, no. 4, pp. 386–398, 2005.

[10] B. Pérez and I. Porres, “An Overall Framework for Reasoning About UML/OCL Models Based on Constraint Logic Programming and MDA,” *Intern.Journal on Advances in SW*, vol. 7, no. 1&2, pp. 370–380, 2014.

[11] S. Sen, B. Baudry, and D. Precup, “Partial Model Completion in Model Driven Engineering using Constraint Logic Programming,” in *Proc. of the International Conference on Applications of Declarative Programming and Knowledge Management (INAP’07)*, 2007, pp. –.

[12] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, “Automatically reasoning about metamodeling,” *Software & Systems Modeling*, pp. 1–15, february, 2013.

[13] B. Pérez and I. Porres, “Reasoning About UML/OCL Models Using Constraint Logic Programming and MDA,” in *Proc. of the Eighth International Conference on Software Engineering Advances (ICSEA’13)*, October 2013, pp. 228–233, ISBN: 978-1-61208-304-9.

[14] FORMULA - Modeling Foundations, Website: http://research.micro–soft.com/en-us/projects/formula/. Last visited on August 2014.

[15] H. Baumann, P. Grassle, and P. Baumann, *UML 2. 0 in Action: A Project-based Tutorial*. Packt Publishing, 2005.

[16] MOFScript Home page, Website: http://www.eclipse.org/gmt/mofscript/. Last visited on August 2014.

[17] The Eclipse UML2 project, website: http://www.eclipse.org/modeling /mdt/?project=uml2. Last visited on August 2014.

[18] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, “UML2Alloy: A Challenging Model Transformation,” in *Proc. of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS’07)*, ser. LNCS, vol. 4735. Springer, 2007, pp. 436–450.

[19] D. Jackson, “Automating first-order relational logic,” *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 6, pp. 130–139, 2000.