UML2ALLOY: A TOOL FOR LIGHTWEIGHT MODELLING OF DISCRETE EVENT SYSTEMS

Behzad Bordbar
School of Computer Science, University of Birmingham
Edgbaston,Birmingham
B15 2TT
United Kingdom
B.Bordbar@cs.bham.ac.uk

Kyriakos Anastasakis
School of Computer Science, University of Birmingham
Edgbaston, Birmingham
B15 2TT
United Kingdom
K.Anastasakis@cs.bham.ac.uk

ABSTRACT

Alloy is a textual language developed by Daniel Jackson and his team at MIT. It is a *formal* language, which has a succinct syntax and allows specification and automatic analysis of a wide variety of systems. On the other hand, the Unified Modelling Language (UML) is a *semi-formal* language, which is accepted by the software engineering community as the defacto standard for modelling, specification and implementation of Object based systems. This paper studies the integration of the UML and Alloy into a single CASE tool, which aims to take advantage of the positive aspect of both the UML and Alloy.

Alloy and UML specification provide two views of the system. In order to synchronise the two views, we make use of the MDA style transformation. In particular, we shall present a Meta Object Facility (MOF) compliant metamodel for Alloy and define a model transformation from the UML metamodel to the Alloy metamodel. Based on the approach presented in the paper, we have implemented a tool called UML2Alloy for the modelling and analysis of Discrete Event Systems. To evaluate the tool, the paper presents a case study involving the modelling and analysis of a prototype manufacturing system.

KEYWORDS

UML, OCL, Alloy, Verification, MDA, Metamodel, transformation

1. INTRODUCTION

The Unified Modelling Language (OMGb 2003) has been widely accepted as the defacto standard for modeling, specification and implementation of Object based systems. One of the reasons for the popularity of the UML, as expressed in the following quote by Warmer and Kleppe (2003), is its semi-formal nature: "*Experience with formal or mathematical notations have led to the following conclusion: The people who can use the notation can express things precisely and unambiguously, but very few people can really understand such a notation.*"

However, the use of formal methods increases the reliability of software systems and enables verification and automated analysis. As a result, the conversion of UML models to formal languages has recently received considerable attention. In particular, there are a number of CASE tools that enable automatic transformation of a model specified in the UML to a formal language. For example Evans et. al. (1999), Roe et. al. (2003), Marcano and Levy (2002) and Delatour and Lamotte (2003) deal with the translation of the UML to Z, Object-Z, B and Petri Nets, respectively. Such tools enable the user to specify the system in UML and conduct the analysis of the system via a formal language. UML2Alloy is a tool for intergrading UML

and Alloy (Jackson 2004) into a single tool as depicted in Fig. 1. Using UML2Alloy, the designer can take advantage of the positive aspects of each modelling language. In particular, the user who is *only* familiar with UML and OCL can use the simulation and analysis facilities provided by the Alloy, while he/she is not required to learn Alloy. The user of UML2Alloy creates a model of the system in the UML. The UML model is automatically transformed into an Alloy model. To verify a statement on the UML model, the user specifies the statement in the OCL (Warmer and Kleppe 2003 ; OMGa 2003) language. The tool transforms a statement from OCL to Alloy and evaluates it on the Alloy model. UML2Alloy, is available for free download at http://www.cs.bham.ac.uk/ bxb/UML2Alloy.html

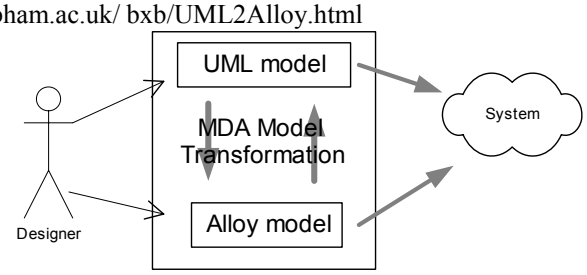

Figure 1. The Outline Of UML2Alloy

Section 2 describes the outline of our approach, which makes use of an MDA (Frankel 2003; Kleppe et. al. 2003) style model transformation from the UML to Alloy. In particular, we shall present a simplified metamodel for Alloy and define a mapping between the UML metamodel and the Alloy metamodel. We shall end section 2 by a brief description of the architecture of UML2Alloy. To evaluate the tool, we shall present a case study involving modelling and analysis of a *bottle capping machine* as a Discrete Event System. Section 3 starts by explaining the example. Section 3.1 presents an outline of a method of the specification of behavioural aspect of DES. Section 3.2 presents a brief explanation of verification via UM2Alloy. The paper ends with a conclusion.

2. MDA MODEL TRANSFORMATION FROM UML TO ALLOY

In the MDA, each model is based on a metamodel that describes its model elements and their relationship (Frankel 2003). In the MDA, model transformation is carried out via defining the transformation rules from a source metamodel to a destination metamodel as depicted in Fig. 2. Transformation rules define a mapping between a source and a destination metamodel. A transformation engine executes the transformation rules on the source model (acting as the input) in order to generate its equivalent destination model (output).

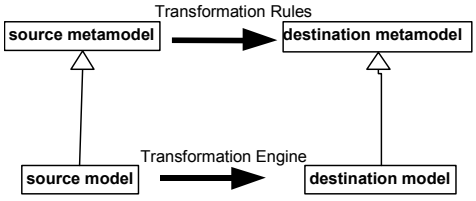

Figure 2. MDA model transformation

Fig. 3 depicts a metamodel for Alloy. An Alloy *Model* consists of one or more *Paragraphs*. There are various types of *Paragraphs* in Alloy. For example, *Signature Declarations* (*SignatureDecl*), which are similar to *Classes* in UML, declare a *Signature* that denotes to a set of atoms (Jackson 2004). A *Signature Declaration* has a *Signature Id* (*SigId*), which is the unique identifier for the Signature. The *Signature Declaration* embodies a *Signature Body* (*SigBody*), which defines *Declarations*, containing a *Declaration Expression* (*DeclarationExp*). A *Declaration Expression* consists of one or more *Expressions*. For further details on Alloy and the meaning of elements in Fig. 3, we refer the reader to the Alloy reference manual (Jackson 2004). Analysis in Alloy is conducted via the discovery of instances of the model or by presenting counter-examples on the *Assertions* about the model. An *Assertion* is a Boolean expression that is true according to *Facts* declared in the model. A *Fact* is an expression about the model that is always valid. Alloy

Analyzer (Alloy), which is a freely available implementation of Alloy, transforms the model into a format that can be evaluated via a SAT solver and uses an off-the-shelf SAT solver for the analysis of the model.

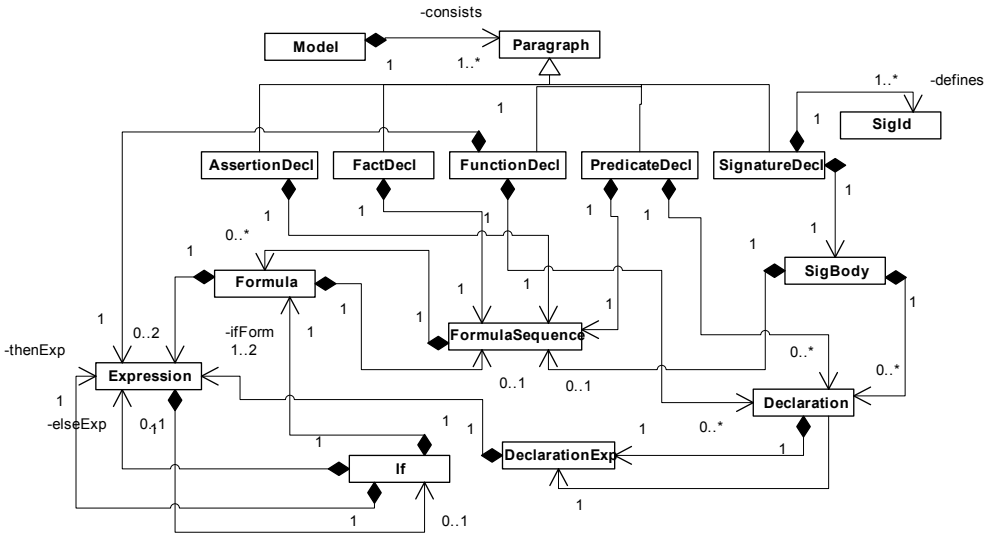

Figure 2. A Metamodel For Alloy

Fig. 4 shows a part of the UML metamodel. The shaded elements represent OCL-related model elements. For further details on UML metamodels and model elements appearing in Fig. 4 we refer the reader to the UML specification (OMGb 2003).

Table 1 depicts the correspondence between the model elements in the UML and Alloy. *Classes* are translated into *Signature Declarations* in Alloy, retaining their original name. *Attributes* of a class are translated into *Relations* on the corresponding *Signature*. Similarly, *Data Types*, which are types that carry pure values, are translated to *Signatures*. *OclExpressions* are translated into Alloy *Expressions*. The OCL expressions such as *If Expressions* are mapped into *If formulas* in Alloy. The translation of the other elements of the Table 1 can be explained similarly. Since the emphasis of Alloy is on modelling aspects of systems, it does not define any notion of types, as in programming languages. However, types could be supported via the use of libraries. For example, *Booleans* are not directly supported but it is possible to use a library that is distributed with Alloy to simulate Boolean data types. In the latest version of Alloy (version 3) there is also basic support for integers.

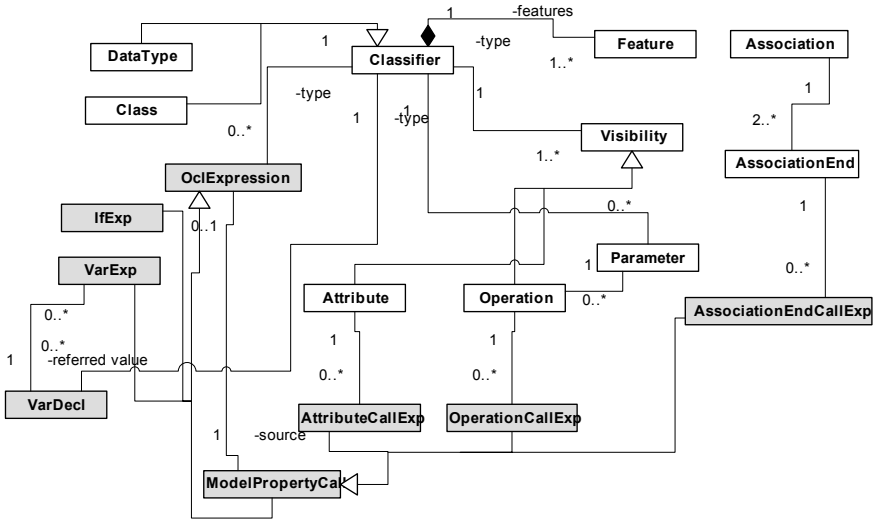

Figure 4. A Part Of The UML Metamodel

We have implemented the aforementioned transformation rules into a tool called UML2Alloy, which is freely available at http://www.cs.bham.ac.uk/~bxb/UML2Alloy.html. Figure 4 depicts the structure of UML2Alloy. The user can interact with the tool either via a UML tool (ArgoUML) or the Alloy Analyzer (Alloy). The "core of UML2Alloy" uses the XML Metadata Interchange (XMI 2003) format generated by the UML tool to transform the model into Alloy.

Table 1. A Subset Of The Transformation Rules

UML	Alloy
Classes	Signature Declarations
Attributes	Relations of the Signature
Data Types	Signature Declarations
OCL Expressions	Formula Expressions
If Expressions	If Formulas
Operations that return a type	Functions
Operations that return void type	Predicates
Operation Parameters	Parameters of Predicates or Functions
Associations	Relations of a Signature

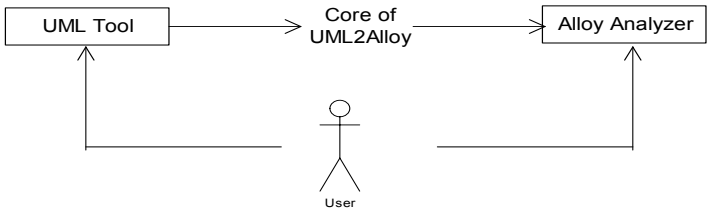

Figure 5. The Structure Of UML2Alloy

3. MODELLING AND ANALYSIS OF A BOTTLE CAPPING MACHINE

The classic paper, Ramadge and Wonham (1989) describes a Discrete Event System (DES) as a "*dynamic system that evolves in accordance with the abrupt occurrence, at possibly unknown irregular intervals, of physical events ..., which requires control and coordination to ensure the orderly flow of events*." There are a variety of DES models. For example, Logical DES, which simplifies the model by ignoring the time of the occurrence of the events and considering the order in which they occur, can be modelled via Petri Nets, Automata, Process Algebra. Logical DES models are used when the model studies the properties of the event dynamic which are independent of timing assumptions. For further details on DES we refer the reader to Cassandras and Lafortune (1999). In this section, we shall study an example of the design of synchronization logic for a prototype bottle capping machine depicted in Fig. 6. The example, which is based on the example studied by Jiang et. al. (1996) is modelled as a Logical DES via UML2Alloy.

Consider a system consisting of two independently driven, independently controlled axes; a *Drum* and a *Slider*. *Bottles* are delivered into the *Drum* from another section of the machine that is marked as *Bottle into system*. The *Drum* which has a rotational movement carries the bottles into the Slider, which caps each bottle. After capping a bottle, it is delivered out of the system from the part which is marked by *Bottle out of system*.

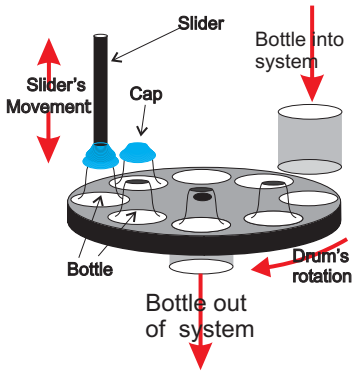

Figure 6. A Prototype Bottle Capping Machine

Fig. 7 depicts a UML model of the systems as a class diagram. The Drum has only one attribute *position,* of type enumeration *d_pos*, which is defined as a separate "*Custom Data Type*" class. The Drum carries the bottles by rotating *D.position = D_Rot* and when a bottle is at a suitable position to be capped, the Drum becomes *stationary D.position = D_Sta*. For the capping to take place, the drum must remain *stationary* (*Stationary Committed*) *D.position = D_Sta_Com*. At this stage, a rotating phase of the drum is completed: *D.position = D_rotate_complete*. Similarly, the Slider has an attribute *position*. A cycle of movement of the Slider starts by its *approaching* to the Drum *S.position = S_App*. To achieve the maximum output of the system, the Slider approaches the Drum with maximum speed. If the slider and Drum are synchronised, i.e. if a bottle is at its suitable position *D.position = D_Sta_Com*, the slider proceeds its movements and caps the bottle. This is modelled as the *S.position = S_insert*. However, if the Slider and the Drum are not synchronised, the slider can *abort* its movement. The decision to *insert* or *abort* is made at the *decision point* (*S.position = S_Dpos*), which considering the momentum of the movement of the Slider, is the last point that the Slider can *abort* its movement. At the decision point, if *insert* is not possible, the Slider *aborts* (*S.position = S_abort*) and waits for the Drum to catch up. Then, when the Drum and Slider are syncronised, the Slider can insert *S.position = S_insert*. Following the insert, the approach phase of the Slider is completed (*S.position = S_approach_complete*). Then, the Drum and the Slider get engaged, i.e. the bottle is capped. This is followed by a reverse movement of the Slider, resulting in the Slider being clear of the drum *S.position = S_cod* (*Clear Of Drum*). At this stage, the motion of the Slider has completed (*S.position = S_motion_complete*).

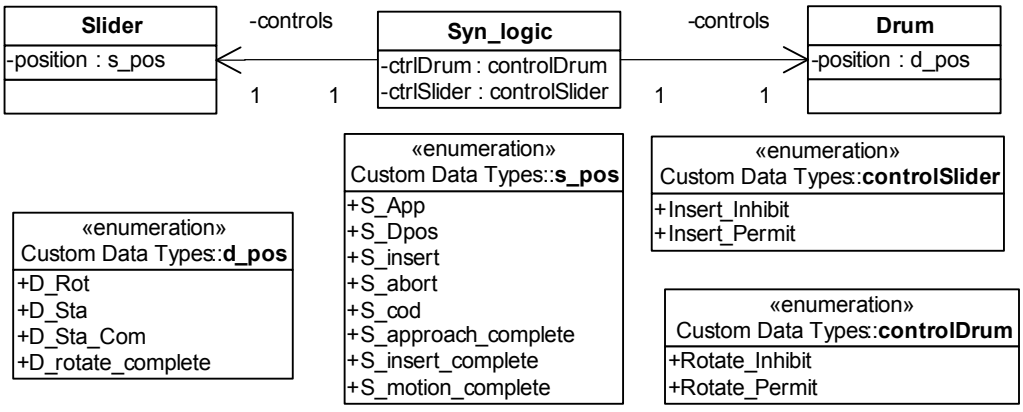

Figure 7. UML Model For The Bottle Capping System, version 1

The Slider and Drum are driven and controlled independently. As a result, the system, which has a highly non-trivial dynamics, requires a component to control and synchronise the Drum and Slider such that the Slider can put a cap on a bottle. This is modelled as the class *Syn Logic*. The synchronisation logic controls the Drum by sending a *Rotate_Inhibit* or *Rotate_Permit* signals to the Drum. Similarly, the *Syn Logic* also

controls the Slider by emitting *Insert_Inhibit* and *Insert_Permit* signals. This is modelled via a pair of attributes *ctrlDrum* and *ctrlSlider* of the *Syn Logic* which are of type *ControlDrum* and *ControlSlider*.

3.1 Modelling the behaviour

Fig. 7 depicts a static view of the system and does not provide any information on the dynamic aspects of the system, which deals with the specification of the evolution of the system from one state to another. For examples, in one of the state of the system, the Drum is stationary (*D.position = D_Sta*). This is followed by the state at which the Drum is rotating (*D. position = D_Rot*). As a result, we shall start by specifying the internal behaviour of the each class via a set of methods. For example the method *D1()* of the Drum which marks the change of state from *rotating* to *stationary*, can be defined via the following OCL statement:

```
context Drum :: d1 ()
pre d1_pre : self . position = # D_Rot
post d1_post : self . position = # D_rotate_complete
```

Similarly, *D2()* defines the change of states from *D_rotate_complete* to *D_Sta* (*stationary*) *D3()* defines the change of states from *D_Sta* to *D_Sta_Com* (*stationary* and *committed*) and *D4()* defines the change of states from *D_Sta_Com* to *D_Rot*. The overall behaviour of the Drum is specified via the function *D_all()*, which is a "motion profile" for the Drum. *D_all()* makes use of the functions *D1()*, *D2()*, *D3()*, *D4()*. Due to space limitations, it is not possible to include the full OCL definition of all methods of the system. The interested reader can find the full set of definitions at http://www.cs.bham.ac.uk/ bxb/UML2Alloy.html.

The next step is to define the collective behaviour of the system, representing the interaction among various components. In the UML (OMGb 2003) such dynamical aspects of the system are often specified via Sequence diagrams and Activity diagrams. A possible approach is to define MDA transformations to translate such models to Alloy. However, defining MDA transformations for mapping behavioural aspect of systems is a highly non-trivial task, which is outside of the scope of the paper. As a result, we shall present a method of specifying behaviour of systems based on *Labelled Transition Systems*. The following present the formal definition of Transitions Systems, which is the underlying semantics for various models of DES such as Petri nets and Automata, see Cassandras and Lafortune (1999).

Definition: A (Label) Transition System is a four tuple (S,A,Δ,s_0) such that S is the set of states of the system, A is a set of *actions* (*events*) that cause a change of state in the system, $\Delta \subseteq S \times A \times S$ defines the transition between two states, by identifying consecutive states. s_0 is the *initial state* of the system.

We model Transition Systems as a part of our models, as depicted in Fig 8 below. We have added a new class *State*. Each object State has instances of the objects of the model. For example, the association *has_slider* shows that a State has one instance of the object slider. The initial state s_0 is identified by the attribute *isInitial*. There are also methods that present previous (*prev*()) and next (*next*()) State(s) of a State. Obviously, the initial State has no *prev* or *next* State, which can be written as an OCL constraint.

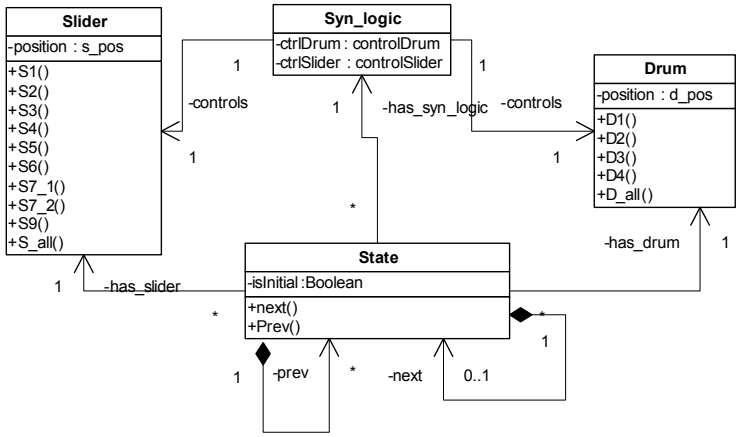

Figure 8. Modelling the behaviour of the Bottle Capping System

State is a reserved Class in UML2Alloy. Like any other class, UML2Alloy transforms the UML class *States* to an Alloy *signature* with the same name. Our implementation makes use of the polymorphic *ordering* module of Alloy, the ordering module distributed with the Alloy Analyzer, which provides support for ordered sets. This is the standard way to achieve process modelling in Alloy (Wallace 2003).

The model depicted in Fig. 8 requires additional constraints, which makes the analysis of the system inefficient. For example, notice that there are two ways to navigate from the class State to the Class Drum. As a result, we need to impose a constraint to specify that the Drum corresponding to a Syn_logic is the same as the Drum to which the State is associated. The same should be defined for the Slider. To avoid imposing such constraint, we have refined the model to associate a State to only the Syn_logic. This way we only have to specify constraints for the association of the Syn_logic with the rest of the model elements in our diagram.

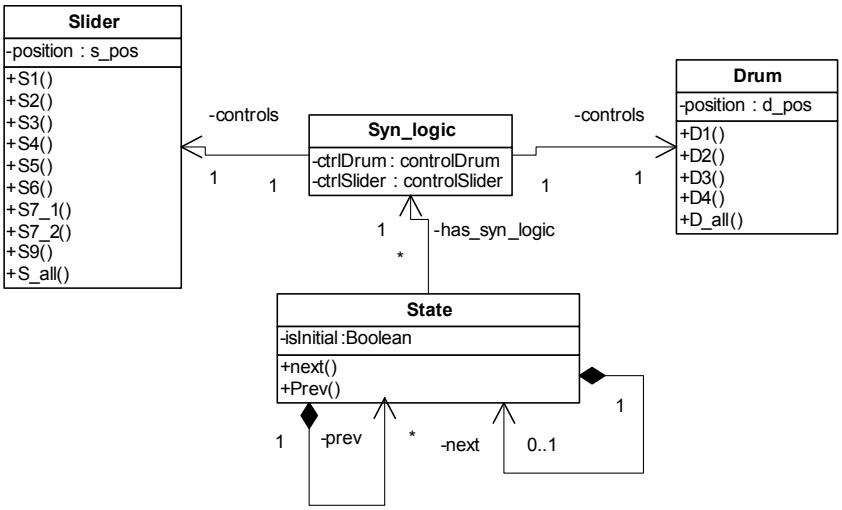

Figure 9. UML Model For The Bottle Capping System

UML2Alloy translates the above model into an Alloy model via using the transformation rules of section 2. For example enumeration types are translated as signatures and enumeration literals are translated into subsignatures which do not have any elements in common.

```
abstract sig dr_Pos{}
sig D_Rot, D_Sta, D_Sta_Com, D_rotate_complete extends dr_Pos{}
```

3.2 Analysis of the model

Using the latest version of the Alloy Analyzer (version 3.0 release candidate 3, which can be downloaded from http://alloy.mit.edu/beta/downloads.php), it is straightforward to simulate the model. However, it can be seen that system has non-trivial dynamics. As a result, it is crusial to analyse the system to ensure its correct functioning. Jiang et al. (1996) present an analysis of the liveness and safety properties of the system via Petri Nets. Using UML2Alloy it is possible to verify such properties with the use of *assertions* in Alloy. For example, we can check the safety criteria that, there is no instance of the model where the Slider inserts the Drum when the Drum is not stationary and committed. This is translated to the following Alloy statement, which means that the *Slider* must not insert the *Drum* when the *ctrlDrum* attribute of the *Syn_logic* has the value of Insert_Inhibit.

```
no s : State | (s.syn.controls_s.position = S_insert) &&
(s.syn.controls_d.position = D_Rot)
```

4. CONCLUSION

This paper makes use of the MDA to define a model transformation from the UML to Alloy. Such transformations can map a UML model into an equivalent Alloy model and facilitate the analysis of the system via Alloy. The paper also outlines a method of specification of behavioural aspects of DES, based on Transition Systems. The method presented in the paper is implemented as CASE tool called UML2Alloy. The paper also presents a case study of modelling and analysis of a prototype bottle capping system via UML2Alloy.

ACKNOWLEDGEMENT

The second author would like to thank his sponsor, the Greek State Scholarships Foundation (I.K.Y.) for the financial support.

REFERENCES

ArgoUML, http://argouml.tigris.org

Alloy Analyzer, http://alloy.mit.edu

Cassandras C. and Lafortune S., 1999. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, London, UK.

Delatour J. and Lamotte F., 2003. ArgoPN: A CASE Tool Merging UML and Petri Nets, *Proceedings of the 1st International Workshop on Validation and Verification of software for Enterprise Information Systems*, *VVEIS*, Angers, France, pp. 94-102

Evans A. et al, 1999. Towards formal reasoning with UML models. *Proceedings of Workshop on Behavioral Semantics*, *OOPSLA'99*. Colorando,USA, pp. 67-73

Frankel D., 2003. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley Publishing, Indiana, USA.

Jackson D., 2004. Alloy 3.0 reference manual, http://alloy.mit.edu/beta/reference-manual.pdf.

Jiang J. et al, 1996. Real-time synchronisation of multiaxis high-speed machines from SFC specification to Petri Net specification. *In IEEE Proceedings – Control Theory and Applications*. volume 143, no. 2, pp. 164-170.

Kleppe A. et al, 2003. *MDA Explained*. Addison-Wesley, Boston, USA.

Marcano R. and Levy N., 2002, Using B formal specifications for analysis and verification of UML/OCL models. *Workshop on Consistency Problems in UML-based Software Development*. Dresden, Germany, pp. 91-105.

OMG, 2003. Object Constraint Language (version 2.0), http://www.omg.org, Document id: ptc/2003-10-14.

OMG, 2003. Unified Modeling Language (version 1.5), http://www.omg.org, Document id: formal/03-03-01.

Ramadge, P. and Wonham W., 1989. The control of discrete event systems. *In Proceedings of the IEEE, Special issue on Dynamics of Discrete Event Systems*. Vol. 77, No. 1, pp. 81-98

Roe D. et al, 2003. Mapping UML Models incorporating OCL Constraints into Object-Z. Technical Report 2003/09, Imperial College, London, UK. http://www.doc.ic.ac.uk/research/technicalreports/DTR03-9.pdf.

XMI, 2003. UML Diagram Interchange final adopted specification, http://www.omg.org, Document id: ptc/03-09-01.

Wallace C., 2003. Using Alloy in process modelling. *In Information and Software Technology*, Vol. 45, No. 15, pp. 1031-1043

Warmer J. and Kleppe A., 2003. *The Object Constraint Language.* 2nd Edition. Addison-Wesley, Reading, USA.