From UML Statecharts to LOTOS: A Semantics Preserving Model Transformation

Valentin Chimisliu
Inst. for Software Technology
Techn. Universität Graz
Graz, Austria
chimisliu@ist.tugraz.at

Christian Schwarzl
AreaE
Virtual Vehicle Competence Center
Graz, Austria
christian.schwarzl@v2c2.at

Bernhard Peischl
Inst. for Software Technology
Techn. Universität Graz
Graz, Austria
peischl@ist.tugraz.at

***Abstract*—A well-founded testing theory encourages the practical application of test case generation techniques. This aims at overcoming the ever increasing complexity of software-enabled systems in the automotive industry. In this article we report on transforming UML Statecharts to the formal language LOTOS. The successful usage of UML Statecharts in our industrial setting and the availability of mature research prototypes for test case generation supporting LOTOS suggest this transformation. Our transformation manages to preserve the semantics of the UML Statechart, allows for treatment of UML-like events, addresses the communication between various models and likewise preserves the atomicity of transitions in the UML Statechart. Moreover, we present first results on the test case generation for our industrial application.**

***Keywords*-LOTOS; UML Statechart; test case generation;**

I. INTRODUCTION

This article reports on an applied research project addressing the introduction of test case generation techniques in the automotive industry. One of the project goals is to establish a scalable test generation technique that allows various test strategies (coverage-based, scenario-based, random, and fault-based). Another goal is to integrate the test generation technique with the existing test automation environment and with the test engineer's domain experience. Thus it is of uttermost importance to rely on familiar and industry-proven diagrammatic notations - in our specific setting this refers to modeling the system behavior by means of UML statcharts.

We have decided to harness the well-founded testing theory behind IOLTS (input-output labeled transition systems) and rely on the mature research prototype TGV [1]. The primary input language of TGV is LOTOS [2] (Language of Temporal Ordering Specification).

In this article we report on the challenges in providing a semantics-preserving model transformation obtained on an industry example. Section II presents our industry example. Section III gives an introduction to the LOTOS language while in section IV we introduce the transformation rules used to derive the LOTOS specification from the UML statechart. Section V addresses model transformation and its challenges. Section VI contains first experimental results obtained by applying a coverage-based test generation strategy on the stated industry example. In section VII we discuss related research and section VIII concludes our paper.

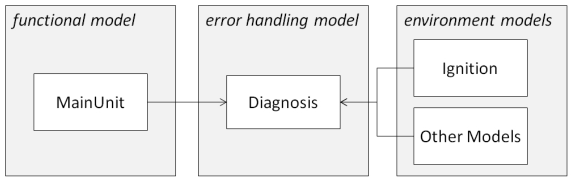

Figure 1. Model Communication Structure

II. MODEL DESCRIPTION

Statechart diagrams are used to visualize state machines and belong to the category of UML behavioral diagrams. As the statechart formalism is well known, we will not go into details on this topic. Detailed descriptions of this type of diagrams can be found in the many books written about UML or in the UML standard [3].

The behavior of the considered system is described by means of UML models [3] representing the diagnosis functionality in modern vehicles. Its purpose is to store the type, occurrence and origin of errors.

Since the diagnosis functionality is distributed over many ECUs, the description of its behavior is also modeled by communicating UML models. The model communication is based on message passing with the possibility to transfer data as message parameters. The models can be grouped (compare figure 1) by the functionality they provide:

1) **Environment Models:** Environment models describe the state of the environment by gathering data from the bus communication or sensors.
2) **Functional Models:** A functional model describes the behavior of an ECU. It comprises the formal specification of the Input/Output communication and error detection routines.
3) **Error Handling Models:** This model defines how a detected error is treated. It specifies which conditions have to be fulfilled to create an entry in the error memory.

Figure 1 illustrates the connections and dependencies between the models. The functionality, which is to be tested in this setting, is given by the Diagnosis model.

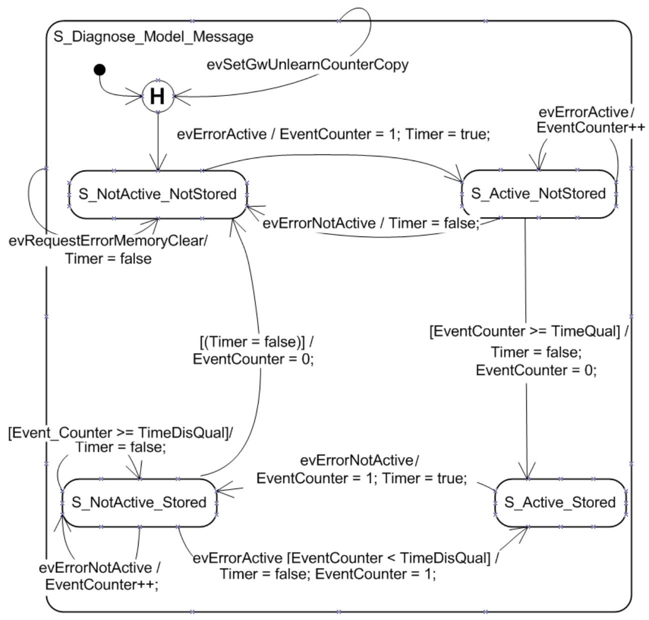
S_Diagnose_Model_Message
evSetGwUnlearnCounterCopy
H
evErrorActive / EventCounter = 1; Timer = true;
evErrorActive / EventCounter++
S_NotActive_NotStored
S_Active_NotStored
evErrorNotActive / Timer = false;
evRequestErrorMemoryClear/ Timer = false
[(Timer = false)] / EventCounter = 0;
[EventCounter >= TimeQual] / Timer = false; EventCounter = 0;
[Event_Counter >= TimeDisQual]/ Timer = false;
evErrorNotActive/ EventCounter = 1; Timer = true;
S_NotActive_Stored
S_Active_Stored
evErrorNotActive / EventCounter++;
evErrorActive [EventCounter < TimeDisQual] / Timer = false; EventCounter = 1;

Figure 2. Diagnosis model

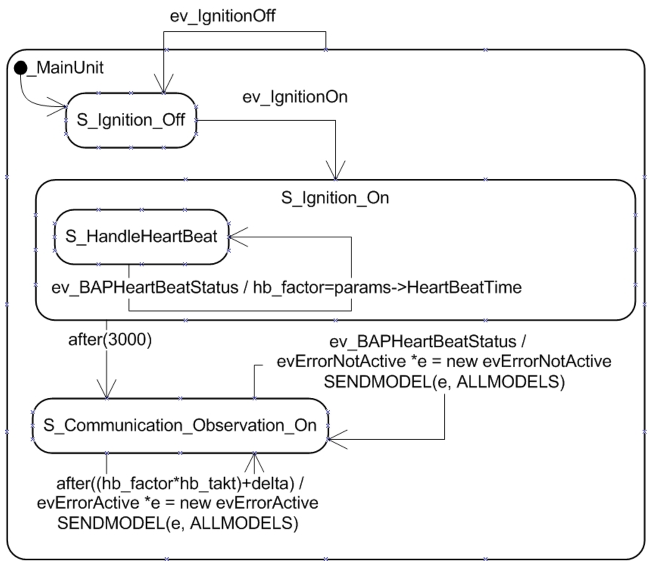
ev_IgnitionOff
MainUnit
ev_IgnitionOn
S_Ignition_Off
S_Ignition_On
S_HandleHeartBeat
ev_BAPHeartBeatStatus / hb_factor=params->HeartBeatTime
after(3000)
ev_BAPHeartBeatStatus / evErrorNotActive *e = new evErrorNotActive SENDMODEL(e, ALLMODELS)
S_Communication_Observation_On
after((hb_factor*hb_takt)+delta) / evErrorActive *e = new evErrorActive SENDMODEL(e, ALLMODELS)

Figure 3. Main unit model

In figure 2 the model of the diagnosis functionality is shown. It consists of five states and accepts four messages namely *evErrorActive*, *evErrorNotActive*, *evRequestErrorMemoryClear* and *evSetGwUnlearnCounterCopy*.

The state *S_NotActive_NotStored* corresponds to normal functioning when no error has been detected. After an error is detected, the system moves to the state *S_Active_NotStored* which means that an error has been detected but is not yet stored. The error is stored after receiving five *evErrorActive* events and the system moves to the *S_ActiveStored* state. This means that the real ECU has now stored an error in its buffer which can be read out with dedicated diagnosis hardware. The diagnosis module shall leave this state and move to the *S_NotActive_Stored* state only after receiving an *evErrorNotActive* event.

Figure 3 specifies the required pre-conditions for recognizing an error in the system behavior: The first condition, which can be identified, states that the ignition has to be switched on for at least three seconds before an error will be detected. When the MainUnit model is in the *S_Communication_Observation_On* state, the message *evBAPHeartBeatStatus* has to be received before the timeout transition is fired and the *evErrorActive* event is sent.

III. LOTOS Introduction

LOTOS [2] is a formal description technique developed within ISO for the formal specification of open distributed systems. LOTOS is composed of an process algebraic part based on Milner's Calculus of Communicating Systems and on Hoare's Communicating Sequential Processes [4], and a data algebraic part based on the abstract data type language ACT ONE.

A LOTOS specification is composed of processes that represent the abstraction of an activity in an implementation. These processes communicate with other processes through a communication mechanism that in LOTOS is termed an event gate (or just gate). This is an abstraction of an interface in an implementation. Below, the structure of a process is presented:

```
process name_proc [gate list] (parameter list) :
            functionality
      behavior
            behavior expression
      where
            type definitions
            process definitions
      endproc
```

Processes are specified through behavior expressions, which represent the state of a process. A predefined set of operators is used to combine behavior expressions to form other behavior expressions. Below some of the most important LOTOS operators are presented:

- The sequentiality operator **";"**, called **action prefix** composes an action g with a behavior expression B to describe a system that will initially accept action g behaving afterwards as B.
- **Choice "[ ]"** composes two alternative behaviors describing a system that offers to the environment two alternatives. An example would be: (Car_started; DRIVE) [ ] (Car_broken; WALK)
- The **full synchronization: "||"** operator denotes the fact that the events which occur in either of the behavior expressions have to synchronize. The expression (a;b;X)||(a;b;Y) may engage in the sequence of events

a;b... .

- The **interleaving** operator **"|||"** allows behaviors to unfold completely independently in parallel. The behavior expression (a;b;c;P) ||| (x;y;T) includes the behaviors: a;x;y;b;c... and a;b;x;c;y... etc.
- **Partial synchronization: "**$|[< gates >]|$**"** means that concurrent behaviors synchronize on the gates listed in the operator. Thus, events occurring at gates in the list will synchronize while the ones occurring at gates not in the list will interleave. The behavioral expression (a;b;c;P) $|[b]|$ (b;y;T) offers the behaviors a;b;y;c... and a;b;c;y... .
- **Inaction: "stop"** represents a system that can not show any action.

IV. Model Transformation

This section describes the transformation rules used to derive a LOTOS specification from an UML statechart description of a system.

A first step of the transformation is that the variables used inside the statechart which are actually the attributes of the class representing the ECU are mapped to LOTOS process parameters while the triggering events are mapped to gates.

A. Transforming Composite States

For each composite state [3] a new LOTOS process is created. For every composite state that does not have a nested history node, the resulting process definition is presented below. The description uses the Extended BackusNaur Form.

process "STATE_ID""["$< gate\_list >$"]"
"("$< param\_list >$"):""**noexit**" ":="
$< substate\_id >$"["$< gate\_list >$"]"
"("$\{< param\_name >\}$")"
endproc

Here $< gate\_list >$ is the list of gates corresponding to the events in the statechart, $< param\_list >$ are the parameters to which the variables in the statechart were mapped to and $< substate\_id >$ is the name of the process generated for the substate targeted by the default transition [3] in the composite state.

For every History pseudostate in the statechart, a new process parameter is inserted in the LOTOS specification. The inserted parameter is used to keep track of the last active state in the statechart. Thus the resulting process definition of a composite state containing a History pseudostate is:

process "STATE_ID""["$< gate\_list >$"]"
"("$< param\_list >$"):""**noexit**" ":="
$< behavior\_expression >$
endproc
$< behavior\_expression >$:=
"["$< hst >$"=" $< s_id >$"] ->"
"$< substate\_id >$"["$< gate\_list >$"]"
"("$\{< param\_name > \}$")"
$\{$ "[ ]" $< behavior\_expression >$ $\}$
$< gate\_list >$:= $\{< gate\_name >$
$\{$"," $< gate\_name >\}\}$
$< param\_list >$:=$< param\_name >$":"
$< param\_type >$
$\{< param\_name >$":"$< param\_type >\}$
$< substate\_id >$:=
"$SUBSTATE\_ID_1$"| ... | "$SUBSTATE\_ID_n$"
$< param\_type >$:="NAT" | "$BOOLEAN$" |...
$< gate\_name >$:="$GATE_1$" | ... | "$GATE_g$"
$< param\_name >$:="$PARAM_1$" |...| "$PARAM_p$"
$< s_{id} >$:="1" | "2" |...| "n"
$< hst >$:="HST_1" | "HST_2" | ... | "HST_m"

In the definition of $< s_{id} >$, n denotes the number of states nested inside the composite state. For simplicity reasons it is assumed that the name of the gate $GATE_i$ also contains the possible parameters of the events in the statechart mapped as gate events.

In figure 2, the state S_Diagnose_Model_Message is a composite state that contains a History node.

Example 1: **process** S_Diagnose_Model_Message
$[gates](parameters\_list)$: noexit:=
[HST1 = 1] $->$ DM_S_NotActive_NotStored
$[gates](parameters)$
[ ]
[HST1 = 2] $->$ DM_S_NotActive_NotStored
$[gates](parameters)$
.....
endproc

In the above example $gates$ is used to represent the gates (events in the original model), $parameters\_list$ is the placeholder for the list containing the names and the data type of the variables used in the statechart, while $parameters$ represent the list containing just the names of the variables (without the data type).

B. Transforming Simple States

Simple states are also mapped to LOTOS processes. Completion transitions [3] shall be fired as soon as their guards evaluate to true.

In order to model this kind of behavior, a conjunction of the negated guards of the completion transitions is added as a guard that restricts the behavior in the resulting LOTOS process. The obtained behavior is that the gates corresponding to non completion transitions are offered to the environment only if the added guard evaluates to true (no completion transition can be fired).

Conflicts between transitions that are fired by the same event and are at different nested levels in the statechart are resolved in a manner similar to the completion transition issue presented above. The structure of the resulting process is presented below:

```
process "STATE_ID""["< gate_list >"]"
        "("< param_list >"):""noexit"":="
{
"[not"< compl_guard >
        {"and" "not"< compl_guard > }"] -> "
"("
< gate_name >"["< ev_guard >
        {"and""not"< ev_guard > }"];"
< state_id >
        "["< gate_list >"]" "("{< param_name > }")"
{"[ ]"
< gate_name >{"["< ev_guard >
        {"and""not"< ev_guard > }"];"}
< state_id >"["< gate_list >"]"
        "("{< param_name > }")"}
")"}
}
{"[ ]"
"["< compl_guard > "] -> "
< gate_name >";"
< state_id >"["< gate_list >"]"
        "("{< param_name > }")" }
endproc
```

The "COMPL_GUARG$_i$" and "EV_GUARD$_i$" in the < *compl_guard* > and < *ev_guard* > defined below represent the guards of the completion transitions and the ones on the other transitions, respectively.

< *compl_guard* >:= "$COMPL\_GUARG_1$" |...| "$COMPL\_GUARD_o$"
< *ev_guard* >:= "EV_GUARD_1"|...|"EV_GUARD_s"

C. Transforming Transitions

When transforming transitions, every direct path between connected states is first constructed. This is done by converting all compound transitions [3] originating from states into transitions that target other states (as opposed to pseudostates).

For every transition originating from a junction pseudostate, a new transition is created containing the triggering event of the transition targeting the junction and the conjunction of the guards of both transition segments (targeting and originating from the junction pseudostate).

In the case of choice nodes [3], the same process applies. The only difference is that the added guard needs to take into account the changes (to the variables) made in the current run to completion step.

The behavioral expression corresponding to a transition has the following form:

```
< trans_behexp > := {< gate_name >}
    {"["< guard >"] ; "} {< gate_name > ";"}
    < state_id >"["< gate_list >"]"
        "("{< param_name > }")"
```

In the expression above the first < *gate_name* > represents the triggering event of a transition and < *guard* > is the mapping of the guard of the transformed transition. The effect part of a transition is represented as follows: if the transition generates an event, this is mapped to the second < *gate_name* > in the above expression, while the operations on the variables in the statechart are mapped to the parameters of the process < *state_id* >.

The state DM_S_NotActive_NotStored from the Diagnose model is transformed into the process:

Example 2: **process** DM_S_NotActive_NotStored

```
        [gates](parameters): noexit :=
  evErorActive ?id:Nat [id = mdl];
  DM_S_Active_NotStored [gates](parameters)
  [ ]
  evReqErrorMemoryClear ?id:Nat[id = mdl];
  DM_S_NotActive_NotStored [gates](parameters)
......
endproc
```

In the proposed transformation the events in the state chart are mapped to gates in the LOTOS specification. The atomic nature of firing transitions is preserved through the fact that once the first action of a transitions is offered to the environment, the only allowed sequence of actions is the one representing the rest of the actions on that transition.

V. Integrating the Environment Model

The presented model architecture assumes a communication between the different ECUs in the vehicle. Even though LOTOS is very well suited to represent communication between processes, there are several aspects that need attention when doing this for the presented example.

A. Preserving the Atomic Nature of Transitions

The actions of a transition are considered as being executed as an atomic unit.

The system is described by communicating processes being synchronized at gates representing the events that are interchanged between the models. In order to preserve the atomic nature of firing a transition, another process that synchronizes on all gates with the already existing processes is added. This process defines the allowed sequence of gate offerings thus restricting the behavior of of the system. An example of such a restriction in the Synchronize process is presented below:

Example 3: **process** Synchronize

```
        [gates](parameters): noexit :=
  ev_BAPHeartBeatStatus ;
  EvErrorNotActive !mdl ;
  Synchronize [gates](parameters)
  [ ]
.....
endproc
```

In the example above the sequence of gates represents the restriction for a transition from the Main Unit model specifying that once the processes have synchronized on gate “ev_BAPHeartBeatStatus” they will synchronize next on gate “EvErrorNotActive”.

B. Treatment of Events

Events that are received but are not handled in the current active state are simply ignored. In the specification this is addressed by making the processes corresponding to simple states input complete. This assumes the insertion in the current process of gates corresponding to the untreated events. When the process synchronizes on such a gate it will simply have no reaction and then call itself.

In the DM_S_NotActive_NotStored process, the choice “evErrorNotActive !DM_trans_0 ?mdl:Nat;” represents such a gate. The gate event “!DM_trans_0” is used to mark a situation when a received event is ignored.

Example 4: **process** DM_S_NotActive_NotStored
    $[gates](parameters)$: noexit :=
......
  EvErrorNotActive !DM_trans_0 ?mdl:Nat;
  DM_S_NotActive_NotStored $[gates](parameters)$
endproc

C. Run to Completion Step

The run to completion step refers to the fact that upon receiving an event, a statechart shall execute until it reaches a stable state; i.e. until it reaches a state where no completion transition can fire.

In the presented example, the number of situations where completion transitions need to be fired is limited. This allows such sequences to be explicitly defined in the synchronization process. A more general solution that allows an automatic treatment of such situations will require the insertion of a queue data structure in order to store the events fired during the current run to completion step. The events stored in the queue can be consumed after the system has reached a stable state.

VI. Experimental Results

One of the goals behind this transformation is to allow for the usage of already existing test generation tools to automatically extract test cases from UML models. One of the most mature tools of this kind is TGV [1] , which is integrated into the CADP toolbox (see http://www.inrialpes.fr/vasy/cadp for details).

TGV uses the concept of test purposes to focus the test case generation and to avoid the state space explosion. The quality of the resulting test cases strongly depends on the skills of the tester - in terms of specifying appropriate test purposes.

Significant effort has been put into the development of techniques to automate the test purpose generation process. Such a technique [5] was used on the Diagnosis model to automatically derive test purposes and subsequently test cases. The results are presented in Table I.

Table I

Coverage Criteria	TestPurpose	Time	Test Cases
Process	5	4,35 sec	5
Action	16	42,07 sec	16

The first column of Table I contains the coverage criterion for which the test cases were generated. The second column presents the number of generated test purposes. In the next column, the total time in seconds needed to process these test purposes is shown. The last column lists the number of generated test cases.

Table II lists the statistics regarding the test cases obtained for process coverage. The first column contains the number of the test case and the second one the number of covered processes (corresponding to the states of the statechart). The next two columns list the number of states and transitions of the IOLTS describing the test case. The last column contains the time needed to compute the test cases.

Table II

No.	Covered processes	States	Transitions	Time
1	2	2	1	0,70 sec
2	3	2	1	0,70 sec
3	3	2	1	0,71 sec
4	4	101	100	0,77 sec
5	5	1120	1119	1,47 sec

When using this method, some of the obtained test cases were very lengthy. For example test case number 5 covering five states of the Diagnosis model ended up having 1120 states and 1119 transitions. One reason for this is the high branching factor of the IOLTS generated from the LOTOS specification and the fact that TGV uses a depth first search algorithm to search the IOLTS.

A proposed solution to reduce the length of the generated test cases is to use a breadth first search algorithm to search the IOLTS. The applied method [5] is based on the insertion of probes into the LOTOS specification. We have used the model checker Evaluator from the CADP toolbox to search the IOLTS using a breadth first search algorithm this way guaranteeing the shortest path to the inserted probe. The results obtained with this approach are summarized in table III while table IV lists the statistics of the test cases generated for process coverage. Thus the IOLTS describing test case no. 5 covering five states ended up having 7 states and 6 transitions.

There are still issues to be resolved regarding the fact that part of the generated test cases contain redundancies (considering the coverage criteria). However, the obtained

Table III

Coverage Criteria	TestPurpose	Time	Test Cases
Process	5	10,13 sec	5
Action	16	30,99 sec	16

Table IV

No.	Covered processes	States	Transitions	Time
1	1	2	1	2,06 sec
2	2	2	1	2,04 sec
3	3	3	2	2,03 sec
4	4	6	5	1,98 sec
5	5	7	6	2,02 sec

results indicate that the proposed model transformation could work in an industrial setting.

VII. Related Work

There have been several approaches to provide a formal semantic to both structural as well as behavioral aspects of the UML [6]. Approaches of how UML statecharts can be verified using the model checker SPIN [7] can be found in [8].

Closer to the technique presented in this paper is the one described in [9], which defines mapping rules for some of the structural and behavioral aspects of UML. Concerning the behavioral aspects, in [9] the focus is on the transformation of activity diagrams to LOTOS. In [10] and [11] transformation rules from a UML statechart to LOTOS are presented. However, the transformation rules do not address issues regarding communicating models. Further limitations [10] are: not allowing the use of data variables in the statechart, considering only normal states and not allowing the crossing of the borders of composite states.

VIII. Conclusions

In this article we report on an applied research project dealing with the introduction of test case generation techniques in the automotive industry. We propose a semantics-preserving model transformation from UML Statecharts to the specification language LOTOS - the primary input language of mature research prototypes for test case generation.

The obtained results indicate that the derived models allow for deducing meaningful test cases with a reasonable computational effort in an automated way. As states in the UML model map to LOTOS processes our transformation allows for coverage-based test case generation.

Future work includes improvements to the transformation algorithm regarding the number of treated components of the statechart formalism.Further research is needed to improve the quality of the generated test purposes in order to eliminate redundancies in the obtained test cases.

Acknowledgement

The authors wish to thank the “COMET K2 Forschungsförderungs-Programm” of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economics and Labour (BMWA), Österreichische Forschungsförderungsgesellschaft mbH (FFG), Das Land Steiermark and Steirische Wirtschaftsförderung (SFG) for their financial support.

References

[1] C. Jard and T. Jéron, “TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems,” *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 297–315, 2005.

[2] ISO, “ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour,” 1989.

[3] OMG, “UML superstructure reference, http://www.omg.org/spec/uml/2.1.2/superstructure/pdf/,” (last visited Jan. 09).

[4] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[5] G. Fraser, M. Weiglhofer, and F. Wotawa, “Coverage based testing with test purposes,” in *Proceedings of the 8th International Conference on Quality Software*, 2008, pp. 199–208.

[6] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner, “Towards a formalization of the Unified Modeling Language,” in *ECOOP’97 – Object-Oriented Programming, 11th European Conference*, ser. LNCS, M. Aksit and S. Matsuoka, Eds., vol. 1241, 1999, pp. 344–366.

[7] G. J. Holzmann, “The model checker SPIN,” *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[8] D. Latella, I. Majzik, and M. Massink, “Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker,” *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, 1999.

[9] P. P. da Silva, “A proposal for a LOTOS-based semantics for UML,” Department of Computer Science, University of Manchester, Manchester, UK, Tech. Rep. UMCS-01-06-1, June 2001.

[10] B. Hnatkowska and Z. Huzar, “Transformation of dynamic aspects of uml models into lotos behaviour expressions,” *International Journal of Applied Mathematics and Computer Science*, vol. 11, no. 2, pp. 537–556, 2001.

[11] R. Mrowka and T. Szmuc, “UML statecharts compositional semantics in lotos,” in *Parallel and Distributed Computing, 2008. ISPDC ’08. International Symposium on*, Washington, DC, USA, 2008, pp. 459–463.