A Semantics for UML Specification to be validated with AGATHA

Céline Bigot	Alain Faivre	Jean-Pierre Gallois	Maria Virginia Aponte	Véronique Donzeau-Gouge
Phd student	Research eng.	Reasearch eng.	Reasearcher	Researcher

DRT/LIST/DTSI/SLA – Bat. 451
CEA-Saclay
F-91 191 Gif sur Yvette Cedex France
Phone: +33 1 69 08 62 58
Fax: +33 1 69 08 20 82
E-mail: {celine.bigot,alain.faivre, jean-pierre.gallois}@cea.fr

CNAM-CEDRIC
292 rue Saint Martin
FR-75141 Paris Cedex 03
Phone: +33 1 40 27 28 15
Fax: +33 1 40 27 22 96
E-mail: {aponte,donzeau}@cnam.fr

Abstract

UML is a modelling language more and more used. However its semantics is not formally defined which arises some problems. These have to be solved when we want to connect some automatic tools as, for example, automatic test generator. Besides the emergence of component notion brings a new research area in the domain of incremental conception and validation.
In a first part, we define a formal semantics for a UML sub-set: this definition allows us to easily derivate a translator compatible with AGATHA, which is an automatic test generator tool. In a second part, we present our current work which takes into account the component notion and uses compositionality properties in the validation domain.

Keywords: formal semantics, validation, component, UML, STGA, LTS.

1. Introduction

Embedded systems integrate an increasing software part which is more and more difficult to control. This part may implement functions concerning both, people and goods safety, or processed data security. In that context, industrials have to use formal methods associated to automated tools, in order to ensure that the obtained results conform to clients attempts. Moreover, due to economical constraints, these methods and tools have to be used right from first phases of the development cycle where the error correction is more advantageous and less expensive.

A tool called AGATHA[1] has been developed at the LSP laboratory (Software for Processed Safety laboratory) of the CEA/LIST (Atomic Research Center / Systems and Technologies Integration Laboratory) since 1996. AGATHA allows to validate specifications via automatic proof of properties and test generation. This tool, based on symbolic execution, can be used on SDL, STATEMATE and ESTELLE specifications. It computes all the symbolic behaviours which can be derived from the system specification, while limiting the combinatory explosion inherent to this type of analysis.

[1] AGATHA: « Atelier de Génération Automatique de Tests Holistiques pour Automates », automatic holistic test generation tool for automata. A series of articles is available.

Another LSP laboratory team has defined a software development methodology, to deal with embedded real-time applications. This methodology, called ACCORD/UML[2], is built on top of a subset of the UML modelling language. It allows analysis, conception and implementation of systems, via an iterative refining process of several intermediate UML models. Conceived as a development-helping tool, this methodology has no validation mechanisms on produced models.

The first aim of our work is to complete the ACCORD/UML methodology with a test-oriented validation tool, via the connection of the two mentionned research works, namely the AGATHA tool and the ACCORD/UML methodology. On one side, any validation approach must start from a precise semantics framework for the validated model, system or language and yet UML lacks of a precise semantics definition. On the other side, AGATHA uses as internal representation the STGA formalism (Symbolic Transition Graph with Assignment) [Lin95], with a precise semantics on which all our associated theoretical works [Lapitre02] are grounded.
In that context, we have defined a formal semantics of operational type based on STGA for the restriction of UML. Herein we present the concrete process used for the definition of the semantics based on a set of rules. These rules enable to quickly derive a safe tool to translate a UML specification into STGA in order to generate tests with AGATHA. With this approach, we aim to deal with some real-time aspects which have to be automatically verified.

The second aim is to give a formal framework to integrate the "component" notion defined in UML2.0. The goal is to offer an incremental validation approach based on components in order to limit test combinatory by means of the compositional properties. In this article, we present the current state of our work.

2. Introduction to the AGATHA toolset

The AGATHA approach, developed at the LLSP aims at helping to achieve formal specification design and validation. Presented specifications are described using automata based formalism. For such specifications, AGATHA generates test cases, from a behavioural graph, obtained by symbolic execution of the specification. These test cases could be carried out during the incremental development process on the specification or on the implementation.

The tool allows different types of model resulting from various formalism like statecharts of the STATEMATE environment [Pierron00], SDL [Lugato01], or ESTELLE [Gallois99]. In this article, we present our current work which allows to add UML as an entry formalism for AGATHA.
The tool translates the initial specification into STGA which is its internal format. This translation allows AGATHA to carry out the symbolic execution of the specification: it permits to obtain a behavioural graph of the specification. Then, thanks to the reduction techniques and with the help of a rewriting tool Brute [Ishisone01], the graph is simplified. Finally, AGATHA uses a constraint solver Omega [Kelly96], which permits to obtain numerical test cases for each path of the graph.

Like every formalism which permits to represent graphs, STGA are composed by nodes and transitions defined as follow.
A **symbolic node** is a pair $n = < s, \{x_1,..,x_n\} >$ where s, is called control node, and represents the label of the node, and, $\{x_1,..,x_n\}$, also noted $fv(n)$, is the set of free variables associated to n. In our case, the set of free variables is the same for every node and corresponds to the set of all variables used in the specification.
Let m and n be two symbolic nodes. A **symbolic transition**, with m source node and n target node, is a tuple $t = < n, b, \alpha, \bar{x} := \bar{e}, n >$ where:

[2] ACCORD/UML is also a part of the AIT-WOODDES project, an European project under the 5th FRDP (Framework Research and Development Program), http://wooddes.intranet.org

- b is a boolean expression,
- α is an action,
- $\bar{x} := \bar{e}$ is a finite set of assignments, with the empty assignment noted θ

α can represent:

- τ the invisible action,
- $c?\bar{x}$ the reception of a vector of values in the vector $\bar{x}$ of variables on the communication channel c,
- $c!\bar{e}$ the emission of a vector $\bar{e}$ of terms on the communication canal c,
- $c?$ et $c!$ the synchronisation on a communication channel c without any exchange of information.

In our case, a **STGA** G is an oriented labelled graph $G = < N, r, T >$ where:

- the set N is a countable set of symbolic nodes,
- r indicates a particular node of N called the root of the graph G,
- the set T is a countable set of symbolic transitions.

To construct the model of a global system, it is natural to design each component of the global system separately. Then, the global model is obtained by setting in co-operation all the components. For the STGA, we use the possibilities offered by the communication actions. A component with a transition $c!\bar{e}$ can send the message $\bar{e}$ to another component which contains a transition $c?\bar{x}$. This emission of message implies an assignment action of the terms $\bar{e}$ to the variables $\bar{x}$ in the receiver transition. Components of the global system which contains transitions with output action can be synchronised with components which contains transitions with input action using the same communication channel.

3. Introduction to the ACCORD/UML methodology

Created by the OMG (Object Management Group), international consortium of industrials and academics, UML (Unified Modelling Language) [UML2.0] is one of the most used object modelling language. It allows to model structure and behaviour of a software system regardless of any method or any programming language.

UML is not a method but a language. The actual version is the 2.0 one. It includes thirteen different types of diagram containing more or less redundant information. If we want to establish some modelling principles to guaranty some properties on the conception process or to restrict UML notations used during conception, a methodology is needed. In our work, we have decided to use the ACCORD/UML methodology [Gerard00] developed in our laboratory. The general process to model an application with the ACCORD/UML approach comprises the three classical steps of development cycle : analyse, conception and implementation. The methodology allows to go from one step of the development to another by an iterative and continuous process of UML refining models.

In a first time, only the behavioural description of the system analysis step is interesting for us, and more particularly state-transition diagrams and activity diagrams.

State-transition diagrams

The ACCORD/UML methodology is limited to protocol state-transition diagrams.
States could be of different forms: initial state, final state, simple state and disjoint composite state (that means state composed by simple states and/or disjoint composite states).
Transitions could have the form :

transition ::= event-name ‘(‘ parameters-list ‘)’ ‘[‘ guard ‘]’
| event-name ‘(‘ parameters-list ‘)’

| signal-name '(' parameters-list ')' '[' guard ']' '/' op-name '(' parameters-list ')'
| '[' guard ']' '/' op-name '(' parameters-list ')'
| '[' guard ']'
| '/' op-name '(' parameters-list ')'

parameters-list ::= parameter | parameter ',' parameters-list

parameter ::= parameter-name ':' type

where: - *event-name* and *op-name* are method names present in the class to which belongs to the state-transition diagram containing the current transition.
- *guard* is a boolean expression on attributes of the considered class and on message parameters just received. This guard has to be satisfied in order to fire the transition.
- *signal-name* is the name of a signal defined in the global model.

The communication semantics between state-transition diagrams is based on a mailbox principle. The choice of the "pushing" and "popping" policy is left to the system designer.

Activity diagrams

In ACCORD/UML, the activity diagram notation can be used on method to describe the set of associated actions.

Different types of state are possible : initial state, final state, action state and send-signal state. In action state, the different possible actions are :

action-state ::= ref-target-object '->' op-name '(' inParameters ')' ';'
| attribute '=' ref-target-object '->' op-name '(' inParameters ')' ';'
| 'RboxRef' '<' type '>' rvalNom ';' ref-target-object '->' op-name '(' rvalName ')' ';'
| 'Return' '(' attribute ')' ';'
| attribute '=' expression ';'

where: - *ref-target-object* is an object name (class or instance name) of the system.
- *op-name* is a method name of the global system.
- *inParameters* are the called parameters of the *op-name* method.
- *attribute* is the set of attributes of the class to which belongs the activity diagram containing the current action.
- *expression* is an arithmetical (+, -, *, /) or a boolean (OR, AND, NOT) expression , on the attributes of the class, on the parameters of the method to which belongs the activity diagram and on constants.

Transitions only allow to condition change from one state to another by a boolean expression.

On state-transition diagrams and activity diagrams some time constraints (delay, timer, …) can be associated to call operation action or to send signal action.

4. Definition of the ACCORD/UML semantics in STGA formalism

Intentionally, the UML semantics has never been precisely defined. Thus this language may allow different semantics for same notations. Within the framework of the ACCORD/UML methodology, several modelling rules have been defined in order to restrict these multiple semantics. However these rules have been written in natural language and are not sufficient to enough restrict UML possible semantics. But precise semantics is necessary to analyse or to state properties on developed models.
Then, as we have seen before, we define a formal semantics for the considered UML subset with STGA in order to automatically generate test cases with AGATHA.

Our semantics is described by translation rules. Each rule has to translate one UML construction in a piece of the whole STGA corresponding to the modelled system. To this aim, in a first part, we define UML constructions and STGA constructions as precise mathematical objects representing the pieces that can compose any model. Once those mathematical objects defined, we can state some well-formedness properties on them. In a second part, we give the rule pattern that we use to define our

translation system. Then in a third part, we present some examples of the different rules that we have produced.

Associated mathematical sets

We start with the description of the mathematical sets that we will need afterwards.

A ACCORD/UML model, in the sense of our work, is composed of a set of signals, global to the model, a set of classes and a set of instances. *Signals* denote the set of all possible signals, *Classes* the set of all possible classes and *Instances* the set of all possible instances.
Likewise, a signal is composed of a name and a set of attributes. The name set is denoted by *SignalNames* and the attributes set by *Attributes*. These sets are defined for the entire system. To have access to the different components of a signal, a set of functions is defined: $f_{Signals\text{-}SignalNames}$ to have access to the signal name and $f_{Signal\text{-}Attributes}$ to have access to the signal attributes.
In the same way, we define all the UML notions that we need for the translation.

The STGA model is expressed with the same form, with a set of STGA, global to the model, and a set of communication channels, also global to the model. We find the notions that we have defined before as the set of symbolic nodes with the set of node names and free variables, the set of symbolic transitions with the set of guards, the set of actions and the set of assignments. Once more, for each of these notions, a global set is defined: for example, *SymbNodes* for the set of symbolic nodes or *SymbTrans* for the set of symbolic transitions. Like for UML, a set of access functions is associated to the sets of all STGA notions.

Pattern of translation rules

We have decided to present our translation system from a UML model to a STGA one with inference rules. In a general manner, an inference rule is represented with a fraction. The conclusion at the denominator is true if the hypothesis at the numerator are verified. We adapt this notation to express ours translation rules :

$$\frac{T \quad P_1 .. P_n}{R \qquad \begin{array}{l} Q_1 \\ \dots \\ Q_n \end{array}}$$

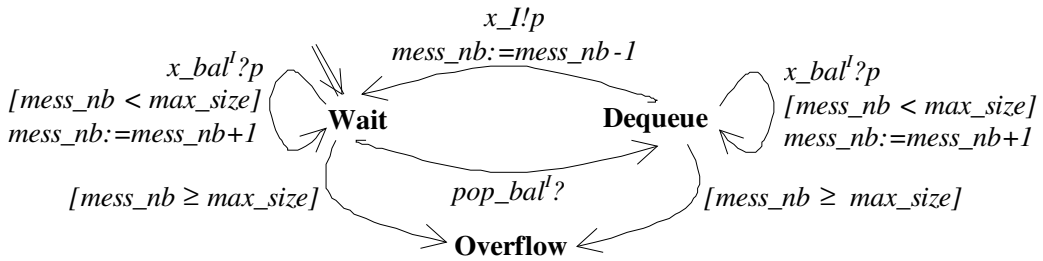
where: - T is the UML element to translate.
- $P_1 \ldots P_n$ are the hypothesis to verify on T.
- R is the STGA translation obtained from T.
- $Q_1 \ldots Q_n$ are properties verified by R.

The translation that we are going to do, is valid for every class instances of the model. To each instance we make a correspondence with a particular STGA.

Communication

As we have seen before, the UML communication is based on the mailbox principle. Every message come through the mailbox. To simulate this notion, which does not exist in the frame of the STGA formalism, we introduce a particular STGA for each class instance. Thus, this particular STGA simulates the mailbox. We introduce two functions to push and to pop messages. But we don't want to say anything about the policy of pushing and popping messages.

The graphic representation of the corresponding STGA is:



Pushing and popping transitions are associated to each method *x* and are prefixed by *x*.

Moreover, the representation of the mailbox principle permits in a first step to abstract some real time constraints on call operation or signal sent. In that case we consider that the time constraints are taken into account by the pushing and popping policy.

State-transition diagrams

The rules allow us to translate elements which constitute state-transition diagrams and activity diagrams.

Thus, for example, for each state *E* of a state-transition diagram , we associate a symbolic node in $STGA_I$, corresponding to the instance *I*, with the property that the name of the node is the name of the state *E*.

For any $I \in Instances$ verifying the following properties:

- $f_{Instances\text{-}Classes}(I) = C$
- $f_{Classes\text{-}StatesDiag}(C) = D$

And for any $E \in f_{StatesDiag\text{-}States}(D)$ we apply the following rule (without numerator):

$$\frac{}{N \in f_{STGA\text{-}SymbNodes}(STGA_I) \qquad \begin{array}{l} f_{SymbNodesNodeNames}(N) = f_{States\text{-}StateNames}(E) \\ f_{SymbNodesVar}(N) = f_{STGA\text{-}Vars}(STGA_I) \end{array}}$$

The case of initial state or final state is subject to particular rules that we don't give here.

For each type of possible transitions we have defined a translation rule, as for states. Thus, for example, for an outgoing transition of an initial state, we obtain the following rules:

- a first one taking into account the case where the method *create*, that permits to fire the transition, hasn't activity diagram,
- a second one considering the case where the method *create* has an activity diagram.

For any transition *T* of the UML state-transition diagram, between an initial state and an unspecified state *S*, triggered on reception of a call to the method *create* having parameters *p* of type *t*, and such as:

- the method *create* hasn't activity diagram,
- and the name of initial state is *init_I*,

then a STGA transition is created which allows to pop a message from the mailbox. A second transition receives parameter *p* of the message *create* on a specific channel, with the following properties:

- the root of the STGA is *init_I* which corresponds to the UML initial state node,
- the intermediate state N_{pop} which allows to pop a message from the mailbox for the UML initial state.

For any $I \in Instances$ verifying the following properties:

- $f_{Instances\text{-}Classes}(I) = C$
- $f_{Classes\text{-}StatesDiag}(C) = D$

And for any $T \in f_{StatesDiag\text{-}Trans}(D)$, we apply the following rule:

$$\frac{T = \bullet \xrightarrow{create(p:t)} S \qquad create \notin dom(f_{Methods\text{-}ActDiag}) \quad (init\_I, \bullet) \in f_{Etats\text{-}NomsEtats}}{init\_I \xrightarrow{true,pop\_bal!,\theta} N_{pop} \in Trans_I \quad f_{STGA\text{-}Root}(STGA_I) = init\_I \quad N_{pop} \xrightarrow{true,create\_I?p,\theta} S \in Trans_I \quad f_{States\text{-}pop}(\bullet) = N_{pop}}$$

Now we can define the rule which allows to translate a transition triggered on reception of a call to the method *create* and having an activity diagram. In that case, we translate the activity diagram into $STGA_I$. When we translate that type of UML transition, we identify the two nodes, N_{Start} and N_{Stop}, where we will connect later the translation of the activity diagram.

For any $I \in Instances$ verifying the following properties:

- $f_{Instances\text{-}Classes}(I) = C$
- $f_{Classes\text{-}StatesDiag}(C) = D$

And for any $T \in f_{StatesDiag\text{-}Trans}(D)$ we apply the following rule:

$$\frac{T = \bullet \xrightarrow{create(p:t)} S \qquad create \in dom(f_{Methods\text{-}ActDiag}) \quad (init\_I, \bullet) \in f_{Etats\text{-}NomsEtats}}{\begin{array}{l} init\_I \xrightarrow{true,pop\_bal!,\theta} N_{pop} \in Trans_I \quad f_{STGA\text{-}Root}(STGA_I) = init\_I \\ N_{pop} \xrightarrow{true,create\_I?p,\theta} N_{Start} \in Trans_I \quad f_{States\text{-}pop}(\bullet) = N_{pop} \\ N_{Stop} \xrightarrow{true,\tau,\theta} S \in Trans_I \quad f_{Activity\text{-}Start}(create, \bullet, S) = N_{Start} \\ \qquad\qquad f_{Activity\text{-}Stop}(create, \bullet, S) = N_{Stop} \end{array}}$$

To adapt different mechanisms defined in the ACCORD/UML methodology, rules have been added, more particularly to manage notions of differed, rejected or ignored messages.

Activity diagrams

If a method has an activity diagram then we translate it in the main STGA of the current instance. So, we need to define translation rules for activity diagrams. The difficulty is that until now the construction of the STGA related primarily to the UML transitions. For an activity diagram the main information is in state. In function of each type of possible action in a state, we define a translation rule from this state to STGA transition(s).

For example, an action can be an assignment of *a* into *x*. Then, we create a STGA transition which allows this assignment. We identify the initial node and the final node of the translation, in order to be able to connect correctly STGA transitions.

For any $I \in Instances$, for any T_{States} verifying the following properties:

- $T_{States} \in f_{StateDiag\text{-}Trans}(f_{Classes\text{-}StatesDiag}(f_{Instances\text{-}Classes}(I)))$
- $f_{Trans\text{-}source}(T_{States}) = e_1$
- $f_{Trans\text{-}target}(T_{States}) = e_2$
- $f_{Label\text{-}Events}(f_{Trans\text{-}Label}(T_{States})) = m$

For any $A \in f_{ActDiag\text{-}States}(f_{Mehtod\text{-}ActDiag}(m))$, we apply the following rule:

$$\frac{f_{EtatsAct\text{-}Actions(A)} = \text{“x=a;”}}{N_{init} \xrightarrow{true,\tau,x:=a} N_{final} \in Trans_I \quad F_{EtatsAct\text{-}source}(A) = N_{init} \quad f_{EtatsAct\text{-}cible}(A) = N_{final}}$$

Thus, we define a particular translation rule for each type of possible action (synchronous operation call, asynchronous operation call) and for each type of particular state (initial state, final state, send signal state). Likewise to each type of possible transition (with or without guard, choice point, …) a translation rule is associated.

Validation of the translation rules
Now that we have defined the translation rules, we want to verify some properties on these rules.

Thus, we want to be able to guarantee that the translation system is complete, that means to each element defined in the initial UML model corresponds at least one element in the corresponding STGA model. This verification is currently made manually and inductively on the rule set. So we will guarantee that every UML elements is translated.

In addition, we want to be able to guarantee that our system is correct, that means all behaviours described in a UML state-transition diagram or in a UML activity diagram are correctly translated in STGA. This verification can't be done in a simple way because the UML semantics is not formally defined. So we can't formally demonstrate that our semantics is correct. But, we can prove that properties on the initial UML model possible to express in our theory are well preserved by induction on the rule set. For example, we can verify that guards associated to every outgoing transitions of a choice point stay in mutual exclusion after translation.

Finally thanks to the rule pattern, we can derivate an automatic translation tool from UML models to STGA ones in a simple manner. Thus, we can use offered functionalities of the AGATHA toolset, in particular automatic test case generation [Lugato02, Bigot03].

5. Current works

Following the growing user's need to structure software applications and to organize them into a hierarchy, a new way to design system, based on object technology, is appeared and has led to a proliferation of associated languages. These languages allow to express the software entity interfaces and to define the corresponding implementation. One goal of the object oriented languages is to improve the software applications' modelling by separating the specification and the implementation levels and optimising their reuse by combining existing classes.
However, the object technology is limited particularly in case of reuse. Indeed, it is practically impossible to know the offered/requested services from the only interface of a class. Furthermore, a class defined in some language can't be directly integrated into an application defined in another language. The methods based on components appeared during last years to correct the defaults inherent in object technology.

The design of applications based on components consists in representing these applications like a gathering of independent software components, interconnected through a communication platform, with the following main motivations:

- a simplified design,
- the reuse of components which reduces software development time and cost,
- the interchangeability of components,
- the possibility to manage together components of different origins whatever their development language and environment.

In this work, our aim is to integrate the component notion in the AGATHA framework. The objective is to define an incremental approach to design a system from its components in order to reduce the combinatory inherent in the symbolic calculus carried out by AGATHA on the overall system.

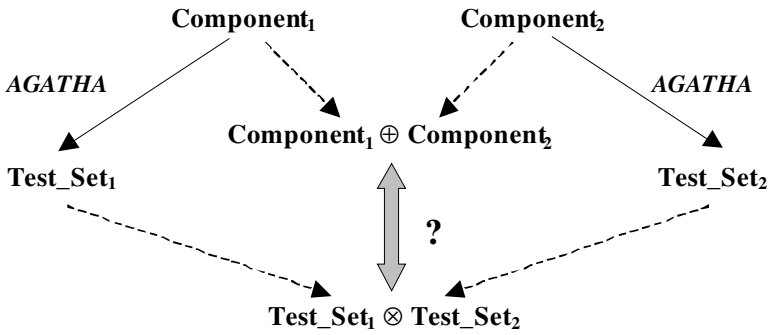
Among others and more precisely, we want to calculate the symbolic execution tree associated to the complete system by composition of the trees associated with each component and previously calculated by AGATHA. Moreover we want to reduce the number of test cases generated for the overall system given that each component has been previously validated by AGATHA.

In a general and informal manner, a component is a software unit which can be easily created, manipulated and installed as a full application. Moreover, it must be independent of the environment in which it is loaded. Thus, each component must have at least one interface which permits to interact with the external environment by receiving and sending information. Components are connected together with ports or connectors which allow information exchanges with different communication mode (synchronous or asynchronous mode) according to their type.

In our case, each component of the system is considered in a "white box" manner as a sub-system, modelled with ACCORD/UML, to which we attach the definition of its communication interface. The transmission of information can be realized by classical method calls and signal exchanges. With each component are associated:

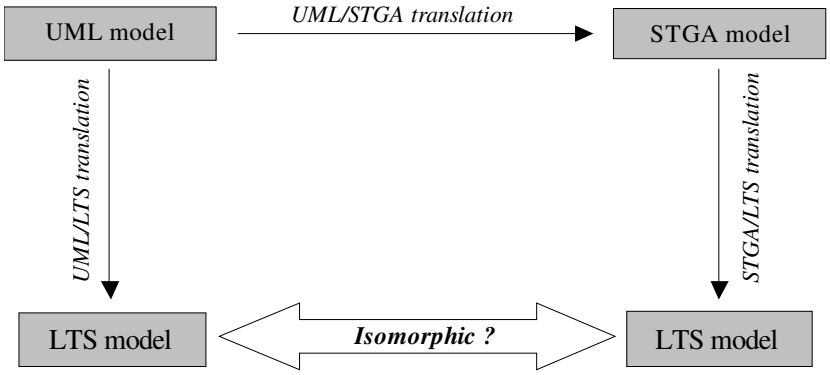
- the list of the supplied interfaces (i.e. the methods of the component which can be called from the external environment and the emitted signals),
- the list of required interfaces (i.e. the external methods which may be called by the component and the signals which may be received by the component).

To study the composition, we need a simple formalism capable of describing the components and with which we will be able to define composition operators for components and resulting test sets. With these operators we want to establish a correspondence between the composition of component models and the composition of test sets associated with each component. The aim of the correspondence is to efficiently use the validation done on each component (i.e. avoid to play again test cases unnecessary to validate the composition), and to add the minimum number of test cases necessary to validate the overall composition. The following figure illustrates all these relations.



The language associated with the Labeled Transition System (LTS) already used in theoretical work of our laboratory [Rapin03], shows such characteristics and had been chosen for the continuation of our study.

Furthermore, the translation of the UML models into LTS will define a denotational semantics for the subset of UML used more formal. As in addition the semantics of the STGA has previously been defined by means of LTS, the comparison of the LTS model directly generated from the UML model with the model generated through the STGA formalism will permit to validate the translation in STGA presented in the first part of this paper.



With this validation process shown in the previous figure, we will try to establish some strong relations, like isomorphism, between LTS models respectively generated from UML model and STGA model.

6. Conclusions

In this article, we have first introduced our process to give a formal semantics to a sub-set of the UML language. With this process, we define a way to verify the completeness and the correctness of this semantics in an operational way. It is interesting to notice that the definition of the formal ACCORD/UML semantics leads us to raise some problems associated to the UML language and to the part of the language used by our methodology. These points have been discussed with the ACCORD team to get rid of ambiguities with respect to the expected semantics. Moreover, the structure of the translation rules allows us to easily derive a tool to translate a UML model into the internal format used by AGATHA in order to generate test cases from this type of specification.

In the second part of the article, we have presented our current work: the introduction of the component notion in AGATHA with the aim of using compositionality properties in the domain of automatic test generation. The validation approach using compositionality seems to be a good way to generate test cases on large systems based on components in order to face the combinatorial explosion problem. Considering the state of the art not very rich in that domain, we have chosen in a first time to deal with this problem by using a very simple formalism with a high abstract level. Then we can concentrate our efforts on the compositionality issue without taking into account the large number of formalisms to define components.

Acknowledgements

The authors would like to thank Christophe Gaston for his help and his constructive comments and suggestions.

Bibliography

[Bigot03] C. Bigot, A. Faivre, J.P. Gallois, A. Lapitre, D. Lugato, J.Y. Pierron, N. Rapin, *Automatic test generation with AGATHA*, TACAS, 7-11 april 2003

[Gallois99] J.P. Gallois, A. Lapitre, P. Lé, *Analyse de spécifications industrielles et génération automatique de tests*, ICSSEA'99, CNAM-Paris, France, December 8-10 1999

[Gerard00] S. Gérard, *Modélisation UML exécutable pour les systèmes embarqués de l'automobile*, Phd Thesis, University of Évry, France, in collaboration with the CEA et PS, October 2000

[Harel87] D. Harel, *Statecharts: a Visual Formalism for Complex Systems*, Science of Computer Programming, vol.8 pp.231-274, 1987

[Ishisone01] M. Ishisone, T. Sawada, *Brute: brute force rewriting engine*, GAIST, January 2001, http://www.theta.theta.ro/cafeobj

[Kelly96] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, *The Omega Library version 1.1.0*, University of Maryland, November 1996, http://www.cs.umd.edu/projects/omega.

[Lin95] M. Hennessy, H. Lin, *Symbolic bisimulations*, Theoretical Computer Science, Vol.138 pp.353-389, Elsevier, 1995

[Lapitre02] A. Lapitre, *Procédure de réduction pour les systèmes à base d'automates communicants : formalisation et mise en œuvre*, Phd Thesis, December 2002

[Lugato01] D. Lugato, N. Rapin, J.P. Gallois, *Verification and tests generation for SDL industrial specifications with the AGATHA toolset*, Proceedings of Workshop on Real-Time Tools, CONCUR'01, Aalborg, Denmark, August 21-24 2001

[Lugato02] D. Lugato, C. Bigot, Y. Valot, *Validation and automatic test generation on UML models : the AGATHA approach*, Workshop FMICS july 2002, Malaga(Spain), ENTCS 66 n°2

[Pierron00] J.Y. Pierron, J.P. Gallois, E. Fievet, A. Lapitre, D. Lugato, *Validation de systèmes industriels par le tests symbolique sur spécifications STATEMATE*, ICSSEA'00, CNAM-Paris, France, December 5-8 2000

[Rapin03] N. Rapin, C. Gaston, A. Lapitre, J.P. Gallois, *Behavioral Unfolding of Formal Specifications Based on Communicating automata*, to appear in Proceedings of the 1st intl workshop ATVA, December 10-13 2003, Taiwan

[UML2] OMG, *Unified Modelling Language 2.0*, OMG, Report formal/2003-04-01, 2003