Automatic Test Generation with AGATHA

Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre,
David Lugato, Jean-Yves Pierron, and Nicolas Rapin

CEA/LIST/DTSI/SLA, CEA Saclay – Bat. 451
91191 Gif sur Yvette Cedex, France
{celine.bigot,alain.faivre,jean-pierre.gallois,
arnault.lapitre,david.lugato,jean-yves.pierron,
nicolas.rapin}@cea.fr

Abstract. This tool demonstration paper describes the AGATHA toolset, developed at CEA/LIST. It is an automated test generator for specifications of communicating concurrent units described using an EIOLTS (Extended Input Output Labeled Transition System) formalism which can be extracted, for example, from UML specification.

1 Introduction

Formal methods allow system analysis and test generation from specifications. This provides an early feedback on a system's behaviour. The economic goal of this specification analysis step is considerable, as it simultaneously reduces cost and time of validation, while increasing system reliability. But these formal techniques are generally quite complex in their use: that is why such techniques have not, at this time, penetrated the industrial domain. One way to decrease this complexity is to provide tools in which the use of those techniques are automated.

This tool demonstration paper describes the AGATHA toolset, developed at CEA/LIST. This toolset is an automated test generator from specifications of communicating concurrent units described using an EIOLTS formalism (Extended Input Output Labeled Transition System). At the present time, AGATHA deals with specifications written in the following languages : UML (the Unified Modeling Language) [1], SDL (Specification and Description Language) [2,3], STATEMATE language [4], ESTELLE [5]. For each of these languages there is a corresponding translator, in the toolset, that transforms the original specification into the EIOLTS language used by AGATHA. Figure 1 shows the main windows of the AGATHA toolset.

The presentation tool will focus on UML specification respecting modelling rules and some UML specializations. These specialization are attached or dedicated to the European project AIT-WOODES [6].

H. Garavel and J. Hatcliff (Eds.): TACAS 2003, LNCS 2619, pp. 591–596, 2003.

2 The AGATHA Kernel

There exist several ways to validate systems specifications. A first one consists in theorem proving and model checking. These kinds of techniques have proved successful for the validation of critical parts of systems. But two major drawbacks to these techniques remain:

1. the combinatorial explosion due to variable domains for the model checking,
2. a need for high-level skills from the developer –who must be aware of formal methods fundamentals– for theorem proving.

Automatic test generation is another way to tackle the problem of systems validation.

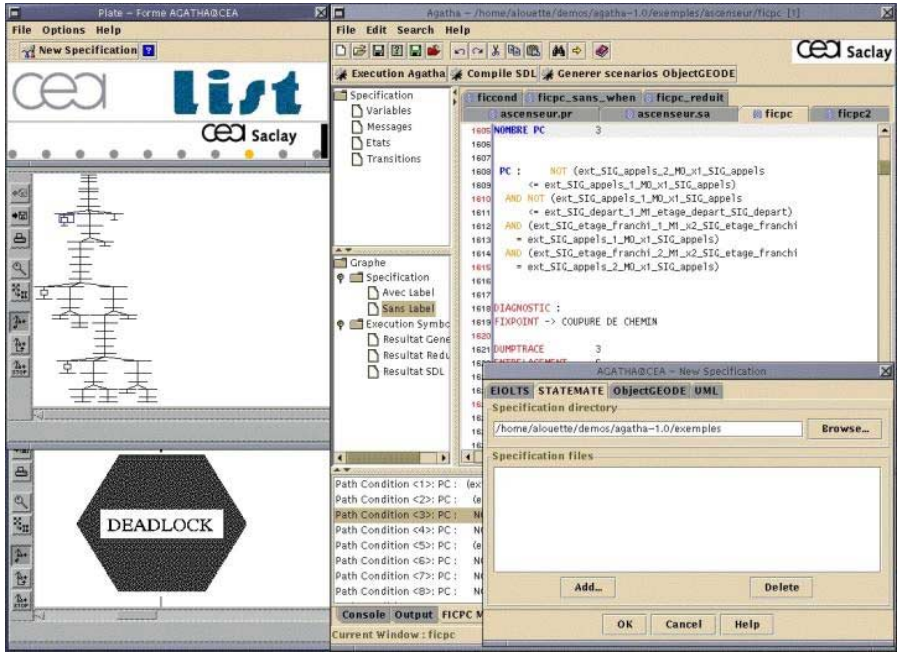

Fig. 1. Main windows of the AGATHA toolset.

The solution adopted in AGATHA is to provide an exhaustive symbolic path coverage. In the future, this criterion will help to use AGATHA for verification. If we want to demonstrate the truthfulness of a property on a specification, because of the exhaustivity obtained with AGATHA, we just have to demonstrate it on the obtained paths.

The following subsections are an overview of the different formal techniques used in AGATHA in order to reach this exhaustive symbolic path coverage.

Main principle: symbolic execution. The major drawback of numeric techniques is the combinatorial explosion due to variable domains. These domains can be huge,

sometimes even infinite. AGATHA uses “symbolic execution” as defined by [7], [8], [9]. Symbolic calculus allows the handling of such domains because computing all the behaviours is not equivalent to trying all the possible values for inputs. Instead of giving values for inputs, they keep their status of symbol all along the execution.

So each behaviour no longer depends on the result of a calculus being completely performed, but on an expression representing constraints on the variables being denoted by the symbols of entries. Each transition fired from a point of the execution adds a new constraint on the variables. At any point of the execution, the entire constraint is called "path condition".

A symbolic state may represent an infinite set of numeric states. The execution tree resulting of the AGATHA computation is a finite tree of symbolic states. The construction of the execution tree is subordinated to reduction procedures in order to eliminate as many redundant paths as possible with different tactics.

A n-tuple of a symbolic node denotes a list of actual control nodes for each of the n concurrent modules. Different heuristics to compute comparison procedures for each symbolic node are also used (inclusion and equality procedures).

Moreover, we currently work on automating several abstraction techniques to reduce complexity and to terminate calculus in any case in order to obtain an exhaustive execution graph.

Simplification procedures. The deeper a point of execution, the bigger the expression representing its path condition. Symbolic expressions of variables may also rapidly grow. That is why a simplification procedure must be applied "on the fly" in order to shorten expressions and detect useless paths.

As of today we use a simplifier based on rewriting techniques. The rewriting engine is Brute [10] that is a part of the CafeOBJ toolset. The rewriting rules file of AGATHA is actually composed of more than three hundred rules. These rules allow both to maintain symbolic expressions within a reasonable size range, and to obtain normal forms for the expressions, easing the comparison between expressions needed in algorithms such as comparison procedures.

We also use a polyhedric tool, Omega [11], in order to compute the inclusion and equality procedures. Using this tool we are able to compare variables domains of two symbolic nodes.

Composition. The symbolic execution process is performed on one module, but the global application is generally composed of many, so they have to be merged.

There are two possible ways to merge modules. The first solution is to use the composition introduced by Milner [12]. The global module is made out of the transitions of its components, except those that are synchronized by a rendezvous. This is due to the fact that we only have communication with rendezvous in the EIOLTS input language of AGATHA. Each rendezvous is replaced by an equivalent transition obtained by eliminating the exchanged parameter. The other solution is to compute the symbolic execution on each module first and then merge the results to obtain the global application behaviour. The major benefit of this latter approach is the parallelization of the calculus: execution trees for each module can be computed separately.

At the moment, only the first solution is implemented in AGATHA. The second option will be integrated soon.

Constraints solvers. Once the execution tree is computed, the whole behaviour of the system is exhibited. Livelocks and deadlocks are visible. We use the DaVinci [13] graphical interface to represent the execution tree. A constraints solver, the Presburger tool Omega, may then be used to get the appropriate values for symbolic variables satisfying path conditions. Then it generates numerical test input sequences. We elect to generate one numeric test for each symbolic test which represents an equivalence class of numeric tests. So the constraints solver computes only one solution for each path condition.

Figure 2 shows the overall architecture of the AGATHA toolset

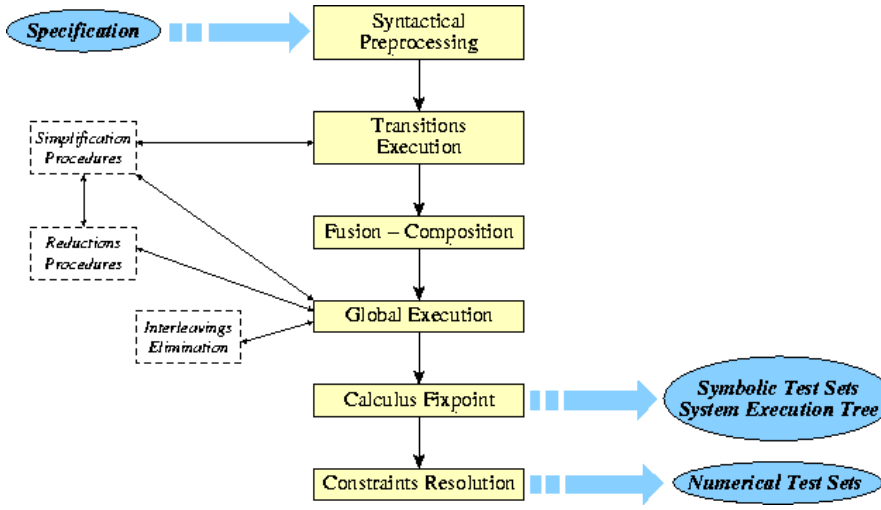

Fig. 2. Architecture of the AGATHA toolset.

3 Transcription of UML Models into EIOLTS

We connect the AGATHA toolset to the environment of the AIT-WOODDES project that offers a method for designing UML specification, an automatic code generator and validation tools. We implement the translation algorithms in the Objecteering 5 UML modeling tool [14]. In this context we generate tests for UML models designed with the ACCORD methodology [15]. The accepted UML models are designed with class diagrams. Each class should have one or more statechart diagram that represents its dynamic behaviour. Collaboration diagrams are used to model interactions between instances of classes. The results provided by AGATHA will be turned into UML sequence diagrams.

The translation from UML to EIOLTS is a two-step process illustrated in Figure 3. First, the UML specification is checked against consistency rules to verify that the

translation modules will be able to translate the specification to EIOLTS; this module also transforms the UML model into an equivalent UML model, only using a restricted set of UML's elements. Secondly, another module translates this restricted UML into an EIOLTS file. The subset of UML that is used is designed to achieve the same level of simplicity in the description of the state machines than the EIOLTS input language of AGATHA.

The generated EIOLTS file is processed by the kernel of AGATHA. Finally, a module analyses the resulting file, translates these results in sequence diagrams and bring these charts back into the Objecteering CASE tool.

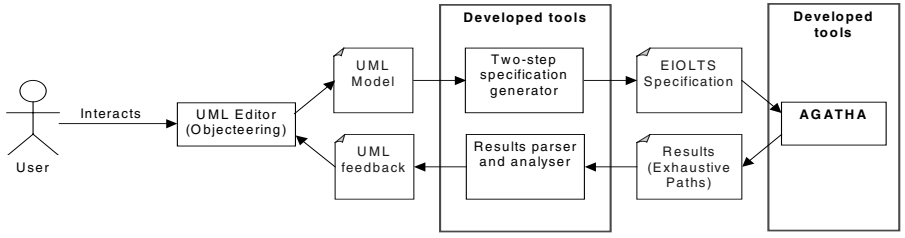

Fig. 3. Translation from UML to EIOLTS.

4 Conclusion

In this tool demonstration paper we have described the AGATHA toolset allowing software developers to validate UML specifications. This toolset may be completely transparent for the user and definitely user-oriented.

Some improvements are foreseen: enriching AGATHA with theorem proving in order to prove properties about the system or connecting an existing model checker to AGATHA. For very large or complex systems AGATHA will also embed new automatic simplification procedures, not working on generated expressions, but on the model itself, and based on abstraction principles. Finally a selection of relevant tests will be performed, along with an estimate of their covering, with respect to criteria or test purposes defined by the user.

References

1. Rumbaugh, I. Jacobson, G. Booch, The Unified Modelling Language Reference Manual, Reading, MA: Addison-Wesley, 1998.
2. Union Internationale des TELECOMMUNICATION, Langage de programmation – Langage de description et de spécification du CCITT – Norme SDL, Recommandation UIT T Z.100, 03/93.

3. D. Lugato, Nicolas Rapin, J.-P. Gallois, Verification and tests generation for SDL industrial specifications with the AGATHA toolset, Proceeding of Workshop on Real-Time Tools, CONCUR'01.
4. D. Harel, Statecharts: a Visual Formalism for Complex Systems, Science of Computer Programming, vol. 8, pp. 231–274, 1987.
5. ISO, *Information processing system, system interconnection, a formal description based on an extended state transition model*, Geneva, 1997.
6. AIT-WOODDES Project N IST-1999-10069, http://wooddes.intranet.gr/.
7. L. A. Clarke. A system to generate test data and symbolically execute programs, IEEE Transactions on software Engineering, vol. SE-2, nº3, September 1976, pp 215–222.
8. J.C. Huang. An approach to program testing, ACM computing surveys.7(3): 113-128, September 1975.
9. J. C. King. Symbolic execution and program testing, Communication of the ACM,19(7). July 1976.
10. M. Ishisone, T. Sawada, Brute: brute force rewriting engine, GAIST, January 2001, http://www.theta.theta.ro/cafeobj.
11. 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.
12. R. Milner. Communication and concurrency, Prentice Hall International, 1989.
13. M. Worner, M. Frohlich, DaVinci Tool version 2.1, Bremen University, July 98, http://www.informatik.uni-bremen.de/davinci.
14. Objecteering Tool version 5, Softeam Paris, 2001, http://www.softeam.fr.
15. S. Gérard, N. S. Voros, C. Koulamas, Efficient system modeling of complex real-time industry; networks using the ACCORD/UML methodology, DIPES 2000.