Instrumental Tool for Automata Based Software Development *UniMod 2*

Dmitry Y. Kochelaev, Bulat S. Khasanzyanov, Bulat R. Yaminov, and Anatoly A. Shalyto

***Abstract*—This article gives a short overview of tool *UniMod 2*, which is now being developed by group of students of Saint-Petersburg State University of Information Technologies, Mechanics and Optics. This instrumental tool is designed to develop automata based software, providing means for visual software building, software verification and validation, as well as means for debugging this software in terms of automata.**

***Index Terms*—Automata, Executable UML, Software debugging, Software verification and validation.**

I. Introduction

INSTRUMENTAL TOOL for Automata Based Software Development *UniMod* [1] year was developed in 2005, it provides means if visual development and execution of automata based software. The base concepts of this instrumental tool are:

- UML [2], [3];
- automata based software development [4];
- Eclipse platform [5];
- Java programming language;
- Open source (sources of the instrumental tool are available on project internet-site [1]).

Actually *UniMod* tool is one of implementations of "Executable *UML*" [6]. When using automata based way the program as a whole is designed by means of two types of *UML*-diagrams:

- Class diagrams are represented as scheme of linked automata based objects;
- State chart diagrams, which implements automata which are present on class diagram.

These diagrams are processed and executed automatically by the instrumental tool, the only thing which is to be developed manually is fragments of code corresponding to input and output actions. Thus this tool let the developer to combine different levels of abstraction (class and state chart diagrams and code on *Java* programming language) and different programming styles (visual and textual).

Give an enumeration of advantages of this instrumental tool:

- The tool provides means of visual building of class and state chart diagrams. The close integration of visual objects representation and their implementation in code is provided as well.
- The tool provides not only the ability to interpret and compile the developed program, yet the ability to perform visual debugging of software, adding breakpoints to states and transitions. Combination of this ability and automata based representation of program make the mistakes search appreciably easier.
- The ability to validate composed automata models provide additional information about possible errors, which were made whereas the state chart diagram was designed.

There are some disadvantages in this instrumental tool as well:

- There is not ability to verify designed programs by means of the tool itself.
- The validation algorithm is not optimal and therefore its execution time could be improved. Moreover the rules are hard coded, that is why it is hard to add new rules to be validated.

II. Development of the Instrumental Tool UniMod 2

The development of second version of the tool *UniMod* was started in 2007. The goal of the project is to eliminate disadvantages, which were present in the first version, as well as improve existing components. One more significant change is closer integration with *Eclipse* platform. This provides an ability to create some parts of the instrumental tool automatically.

A. Data Model

As it was already mentioned one of the primary differences in instrumental tool *UniMod 2* from the prototype is widened use of resources and means, provided by *Eclipse* platform. This let to improve all components of the system. In particular *EMF* (*Eclipse Modeling Framework*) [8] was used for description of the meta model. Meta model is also a model which describes the structure and behavior of other models, which corresponds to this meta model. The *EMF* project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in *XMI*, *EMF* provides

Manuscript submitted March 31, 2008.

D. Y. Kochelaev is with Saint-Petersburg State University of Information Technologies, Mechanics and Optics, Saint-Petersburg, Russia (corresponding author to provide phone: +7-921-919-78-26; e-mail: dkochelaev@gmail.com).

B. S. Khasanzyanov is with Saint-Petersburg State University of Information Technologies, Mechanics and Optics, Saint-Petersburg, Russia (e-mail: bulemar@gmail.com).

B. R. Yaminov is with Saint-Petersburg State University of Information Technologies, Mechanics and Optics, Saint-Petersburg, Russia (e-mail: bulat.yaminov@gmail.com).

A. A. Shalyto is with Saint-Petersburg State University of Information Technologies, Mechanics and Optics, Saint-Petersburg, Russia (e-mail: shalyto@mail.ifmo.ru)

tools and runtime support to produce a set of *Java* classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Utilization of this framework saves time with automatic classes generation, as well as helps to avoid errors, which could be done if these classes were developed manually. Model alteration (e.g. adding new attributes to model items) is performed in visual editor and doesn't require manual code modification, since it will be rebuild automatically in the future.

B. Validation

New algorithm of validation of automata based programs [9] is used in new version of the instrumental tool *UniMod*, which is currently being developed. While validation the model is being checked to suit the meta model. The new algorithm is faster then the one which was used in original version of the tool. This was achieved with utilization of preliminary calculations, as well as utilization of context-dependent rules checking – the model rules are checked only in context of modified model element. This significantly reduces the count of rules, which should be checked, and increases validation speed.

The model validation algorithm consist of two linked stages:

1. On the first stage (before editing the model) the tool collects data, which is required to determine quickly, which may have been violated while editing model.
2. The set of rules is created on each model modification. This is the set of rules, which it is necessary to check.

Rules, which describe the restrictions on the model (actually these restrictions describe the meta model), are stored in *Object Constraint Language* (*OCL*) [10]. This language was developed by *IBM* corporation specially to describe restrictions on *UML*-models [2], [3], and this can be used for solving validation problem.

To measure productivity of the described algorithm it was compared theoretically and experimentally with the original validation algorithm, which was used in UniMod. As a result it was proved that proposed algorithm is much faster comparing to the original one [9].

C. Verification

A new feature of instrumental tool *Unimod 2* is the ability to verify correctness of developed automata programs [11]. To do this verification tool *Bogor* [12] was integrated with *Unimod 2*. As a result, automata programs developed using instrumental tool *Unimod 2* can be checked for correctness by established *Model Checking* [13] method.

Verification method implemented in instrumental tool *Unimod 2* has an advantage over other methods. To verify a program by *Model Checking* method, a special kind of formal model must be constructed, which represents the program. Statements to verify also have to be translated from terms of program to terms of the constructed formal model. Therefore there used to be two similar ways to verify automata programs. The first one [14] was to convert automata program into Kripke model [13], and then verify this manually by standard model checking algorithm. Another way [15] was to translate automata program into an input language of a verifying tool, and then run this tool to check the correctness of the model. If automata program is not correct, an error report is returned by verifying tool or by verification algorithm. However, the error report describes the program incorrectness in terms of the *converted* formal model, not the initial automata program. So user has to convert the error report back into terms of initial automata program, which sometimes is quite difficult.

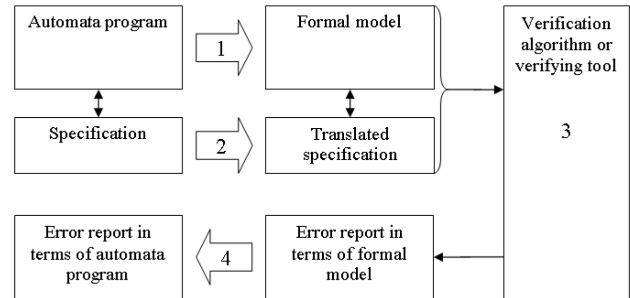

Fig. 1. Previous verification method. Four steps are required: (1) translate the program into formal model, (2) translate specification into formal mode specification, (3) verify the model and (4) translate error report back into terms of automata program.

Fig. 1 represents schema of described ways to verify automata programs.

The advantage of new verification method implemented in *Unimod 2* is that it lets verification tool *Bogor* work directly with automata program, as if it was already a formal model, ready for verification. As a result, there is no need in complex translations of automata program to formal model and back. Therefore, verification process is simpler, faster and more reliable. Schema of this verification method is represented by Fig. 2.

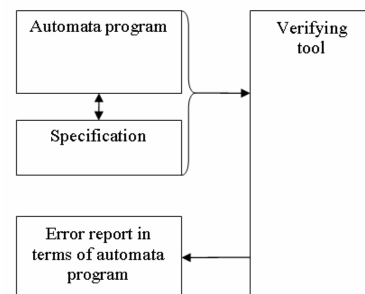

Fig. 2. New verification method: no additional converting are needed.

Actually *Unimod 2* became the first tool which lets create automata based programs, launch, and *automatically* verify them. To check correctness of an automata program, user has just to print a list of statements, which must hold true for the program, and launch the verification. When verification is completed, user is informed which statements do hold or do not. Furthermore, if a statement is violated, user gets a detailed history of program execution, which caused the statement violation. It lets user easily trace down the root cause of the error, and it is one of the main advantages of

program verification over program testing. Another advantage is that verification checks *all* possible program executions to find an error, while each test checks only one execution

D. Debugging

The new concept was used to create debugging part of the instrumental tool *UniMod 2*. The debugger for domain-oriented programming language (e.g. automata language) was created with utilization of *EMF* technology [16]. Utilizing the proposed method it is possible to implement universal debugging system for arbitrary domain-oriented programming language, which implements standard available in *Eclipse* IDE debugging tools:

1. breakpoints give ability to pause program execution in specified place (e.g. on some transition or in some state);
2. step by step execution;
3. overview of context variable in any moment of program execution.

E. Visualization

The *Graphical Modeling Framework* (*GMF)* [17] was used to create model editor, which is used for model editing, as well as for visualization and control of debugging process. The GMF provides a generative component and runtime infrastructure for developing graphical editors based on *EMF*. So this technology provides an ability to build fully functional graphical model editor basing on meta model description and some additional configuration descriptions. This editor can easily be integrated into *Eclipse* IDE. Additional configurations specifies the list of meta model elements, which can be added to diagram, as well as their graphical representation and sets of connected to them editor control elements.

There is an ability to highlight some model elements in the editor [16]. This ability is used to highlight violated constraints, which were found while validation process [9]. This functionality will also be used to visualize results of program verification. Thus an unified approach for visualization actions on model (debugging), model changes and highlighting of model elements (validation) was designed and implemented in the instrumental tool *UniMod 2*.

III. Conclusion

Following components were redesigned, improved or introduced as a result of development of the instrumental tool *UniMod 2*:

1. the representation of automata model was redesigned to improve usability of model editing facilities;
2. the algorithm of automata program validation was improved for better performance;
3. the ability to verify automata programs was introduced;
4. the visual model editor was redesigned to be improve integration with editing tools provided with *GMF* technology;
5. the debugging system for domain-oriented programming language (adopted for debugging of automata models) was introduced;
6. the unified system of debugging and validation visualization was introduced.

References

[1] *V. S. Gurov, M. A. Mazin, A. S. Narvsky, A. A. Shalyto* Instrumentalnoe sredstvo dlya podderzhki avtomatnogo programmirovaniya // Programmirovanie. 2007. № 6, pp. 65–80. Available: http://is.ifmo.ru/works/_2008_01_27_gurov.pdf

[2] *H. Goma.* Designing concurrent, Distributed, and Real-time Applications with UML; Addison-Wesley – 2000

[3] *I. Jacobson; G. Booch; J. Rumbaugh* (1998). The Unified Software Development Process. Addison Wesley Longman.

[4] *A. A. Shalyto*. Switch-tekhnologiia: algoritmizatsiia i programmirovanie zadach logicheskogo upravleniia. Sankt-Peterburg: Nauka, 1998. Available: http://is.ifmo.ru/books/switch/6

[5] Web-site of Eclipse project: http://www.eclipse.org/

[6] *Mellor S., Balcer M.* Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, 2002.

[7] *N. I. Polikarpova, A. A. Shalyto.*Uchebno-metodicheskoe posobie po distsipline «Avtomatnoe programmirovanie». Spb.: SpbGU ITMO. 2007 Available: http://is.ifmo.ru/books/_umk.pdf

[8] *Web-site of Eclipse Modeling Framework project:* http://www.eclipse.org/modeling/emf/

[9] *D. Y. Kochelaev* Metody dinamicheskoy proverki prvil neprotivorechivosti avtomatnoy modeli. Bakalavrskaya rabota. Spb.: SpbGU ITMO. 2007. Available http://is.ifmo.ru/papers/_kochelaev-bachelor.pdf

[10] *Warmer J., Kleppe A.* The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, 2003.

[11] *V. S. Gurov, A. A. Shalyto, B. R. Yaminov.* Technologiya verifikatsii avtomatnyh modeley programm bez ih tranlyatsii vo vhodnoy yazyk verifikatora // Materialy mezhdunarodnoy nauchno-tehnicheskoy konferentsii «Mnogoprotsesornye vychislitelnye i upravlyauschie sistemy». T. 1, pp. 198–203. 2007. Available: http://is.ifmo.ru/verification/_jaminov.pdf

[12] *Robby, Dwyer M., Hatcliff J.* Bogor: A Flexible Framework for Creating Software Model Checkers /IEEE Conf. of the Testing: Academic & Industrial Conference – Practice and Research Techniques (TAIC PART) 2006, pp. 3–22. Available: http://ieeexplore.ieee.org/iel5/11139/35654/01691665.pdf?arnumber=1691665

[13] *E. M. Clarke, Jr., O. Grumberg, D. A. Peled*, Model Checking, MIT Press, 1999.

[14] S. E. Velder, A. A. Shalyto. Verifikatsiya prostyh avtomatnyh programm na osnove metoda Model checking //Materialy XV nauchno-metodicheskoy konferentsii «Vysokie intellektualnye technologii i innovatsii v obrazovanii i nauke». SpbGPU. 2008, pp. 285–288. Available: http://is.ifmo.ru/download/2008-02-25_politech_verification.pdf

[15] *M. A. Lukin, A. A. Shalyto.* Avtomaizatsiya verifikatsii visualnyh avtomatnyh programm //Materialy XV nauchno-metodicheskoy konferentsii «Vysokie intellektualnye technologii i innovatsii v obrazovanii i nauke». SpbGPU. 2008, pp. 296, 297. Available: http://is.ifmo.ru/download/2008-02-25_politech_tezis.pdf

[16] B. S. Khasanzyanov. Metod sozdaniya otladchikov dlya domenno-orientirovannyh yazykov programmirovaniya na osnove

technologii Eclipse Modeling. Bakalavrskaya rabota. Spb.: SpbGU ITMO. 2007. Available:

http://is.ifmo.ru/papers/domainlanguagedebugmethod/

[17] *Web-site of Eclipse GMF project.* http://www.eclipse.org/modeling/gmf/