

Using ASM Specifications for Compiler Testing

A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov

Institute for System Programming of Russian Academy of Sciences
{ka,kos,petrenko,posypkin,vova}@ispras.ru

1 Introduction

Developing an adequate set of tests also called a **test suite** is an important part of software development process. We faced this problem while working on mpC parallel programming language [7] compiler. The general task was to develop a test suite for checking whether the particular compiler implementation correctly processes the programming language.

In this case study we focus on testing language expressions. mpC provides powerful operators for array-based and parallel computations. That is why mpC expressions are complicated and difficult to implement and require thorough testing. However the proposed techniques is also applicable to other parts of the language.

Our approach is a sort of "specification-based testing" [9, 10, 1]. We use Abstract State Machines [4] formalism for modeling mpC expressions semantics. The formal specification is implemented using the ASM-based Montages framework [6] – a new method for giving the semantics of a programming language.

We use the specification for three different purposes:

- **Test cases generating.** mpC specification consists of several Montages. Each montage defines the semantics of a particular abstract syntax tree node. Test programs are generated by combining abstract syntax tree nodes. Incorrect tests are filtered out by the specification. Correct tests constitute the test suite.
- **Test oracle generating.** Executable specification is used for generating trustable output of the given test program. Test oracle compares actual and trustable outputs for a particular test. If results are not identical the verdict is failure.
- **Providing test coverage criteria.** Analysis of the specification coverage allows one to see whether all specification rules were involved while executing the test suite. If the coverage criteria are satisfied then no more test cases are needed, otherwise additional test programs should be added to the test suite.

The paper is organized as follows. Section 2 explains how mpC expressions semantics was defined using Montages. Section 3 overviews the test generating process. Practical results and future work are discussed in sections 4 and 5 respectively.

2 The ASM Specification for mpC Expressions

Overview of the mpC Language. mpC is a parallel programming language supporting computations on a variety of parallel platforms ranging from local area networks to high performance supercomputers. mpC language is a strict two-level ANSI C extension.

First level (also called C[]) supports array-based computations in a spirit of FORTRAN 90 [8]. The language introduces special operators for manipulating arrays as a whole.

mpC extends C[] with facilities for parallel computations. mpC expressions could be used for expressing both computations and data exchange between different computing nodes. More information could be found at [7] or www.ispras.ru/~mpc web site.

So expressions in mpC have much more sophisticated semantics than in C language. Thus the part of the mpC compiler implementing expressions is rather complex and require thorough testing.

Montages. Montages [6] are a semi-visual formalism for describing programming language syntax, static and dynamic semantics. Montages have been successfully used for the specification SQL [3], C [5] and other programming languages.

A language specification is given as a collection of Montages, each of which is associated with a production rule. A Montage consists of a production rule, static semantics rules, condition, dynamic semantics rules and a control-flow graph. Condition, static and dynamic semantics rules are written in XASM [2]: ASM-based programming language. Sample Montage for assignment operator is demonstrated on Fig. 1.

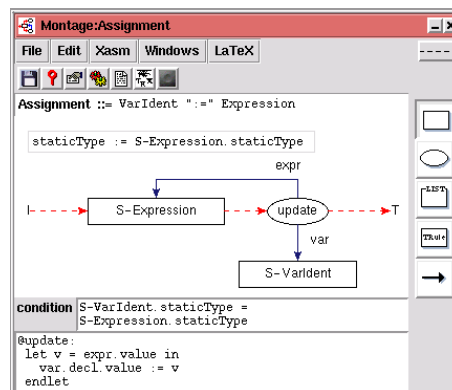


Fig. 1. Montage for assignment operator.

The specification of mpC expression semantics was implemented using Gem-Mex – a tool for developing Montages based specifications. The tool produces executable module implementing the interpreter for the specified language.

3 Generating the Test Suite from the Specification

The Test Suite Architecture. The test suite consists of mpC programs accomplished with their trustable outputs. A test program contains several initializations of variables involving in testing expression, testing expression itself and the "printf" function call for outputting the expression value.

The test suite run is organized as follows. First every program from the test suite is compiled by the compiler under test. Second obtained binary file is executed to produce **actual output**. Third actual output is compared with trustable one. If outputs are not identical the verdict is failure.

The Test Cases Generating Scheme. The proposed scheme of test suite generating is depicted at Fig. 2. We omit some technical details in order to make explanation clear and concise.

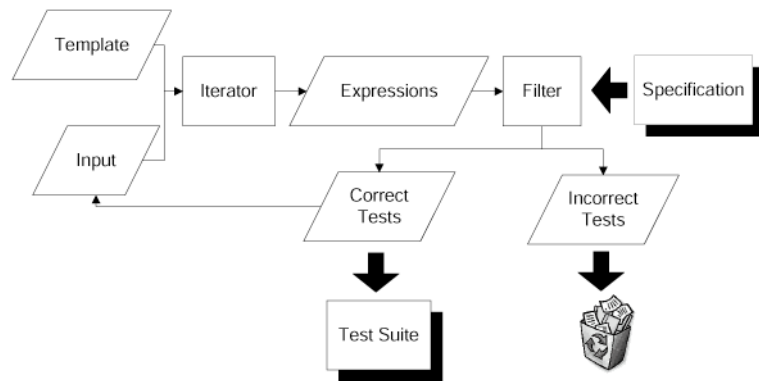


Fig. 2. The scheme of the test suite generating.

The **Iterator** produces syntactically correct mpC expressions. Then produced expressions are processed by the Montages specification and expressions violating semantics constraints are filtered out.

Each expression from the generated set is used for constructing corresponding test program by accomplishing it with initialization and output parts. Obtained test program is processed by the specification in order to produce trustable output.

The **Iterator** generates test programs from two files: **Template** and **Input**. **Template** is a set of several mpC operators. **Input** is a set of several mpC

expressions. The generating is implemented as a substitution of operands from **Input** as operands of operators from **Template**.

Initially **Input** consists of basic expressions like constants and identifiers. Therefore first step of the generating produces only expressions containing one operator. At the second step expressions generated at the first step are used as an **Input** thus allowing to generate expressions containing two operators. The third step uses expressions generated at the second step and so on.

The natural question arises: when to stop? Intuitively it is clear that first step (expressions with only one operator) is not enough. Obvious approach in testing is to use coverage-based heuristics to measure the test suite quality. We consider the coverage of the specification in order to provide implementation-independent test suite adequacy criterion.

For the moment we use update rule coverage [1] to check whether every update rule in static and dynamic semantics parts of each Montage is exercised. Experimental evaluation shows that second step produces the test suite satisfying this coverage criterion.

4 Practical Results

The proposed technique has been successfully applied to mpC compiler testing. The initial **Template** contains all mpC operators. The first and second steps of the test suite generating produces 135 and 13473 test cases respectively. The following table presents the results of testing for both test suites:

	1st Step	2nd Step
No Errors	47	1007
Static Semantics	51	7271
Code Generating	30	3995
Segmentation Fault	6	1138
Result Mismatch	1	60
Run-time Error	0	2

Analysis of failed tests shows that there are 11 distinct errors in the compiler under test. Step 2 introduces a new kind of errors (run-time error) and new errors of existing kinds. This confirms the intuitive idea that test suite consisting of expressions with only one operator is not sufficient for comprehensive testing.

5 Future Work

The bottleneck of the proposed technique is a huge amount of tests. For example in our case second step consumed 63 hours on a 1GHz Linux workstation. The estimated time for the third step is approximately one year. The main time-consuming part of the test suite generating is a run of the specification for filtering out incorrect test cases.

For typical language a very small percentage of syntactically correct programs are also semantically correct. Thus we can significantly reduce the time of tests generating by providing more "intelligent" **Iterator** producing less amount of semantically incorrect tests. Currently we are working on more complex **Iterator** relying not only on syntactic but on semantics structure of the language also.

Another direction of future research is developing more elaborate coverage notion for Montage specification. For the moment we use update rule coverage – a weakest of all possible coverage measures for ASM-based specification. We plan to consider other coverage measures for both dynamic and static semantics parts of the specification.

Since Montages specifications are based on BNF representation of the language syntax from one hand and ASM specification of the language semantics it seems to be reasonable to combine grammar coverage and ASM-coverage metrics to obtain integral coverage measure.

Besides testing compiler on correct input it is very important to check whether compiler processes semantics errors properly, i.e. generates adequate error report. We plan to develop an efficient technique for handling not only correct but also incorrect test cases which are filtered out for the moment (see Fig. 2).

Acknowledgments. We would like to thank Philipp Kutter and Mathias Anlauff for excellent assistance with Gem-Mex tool.

References

1. A. Gargantini and E. Riccobene. ASM-based Testing: coverage criteria and automatic tests generation. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 262–265, February 2001.
2. M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000.
3. B. DiFranco. Specification of ISO SQL using Montages. Master's thesis, Università di l'Aquila, 1997.
4. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
5. J. Huggins and W. Shen. The Static and Dynamic Semantics of C. Technical Report CPSC-2000-4, Kettering University, Computer Science Program, 2000.
6. P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
7. Alexey Lastovetsky, Dmitrij Arapov, Alexey Kalinov, and Ilya Ledovskih. A parallel language and its programming system for heterogeneous networks. *Concurrency: Practice and Experience*, 12:1317 – 1343, 2000.
8. M. Metcalf and J. Ried. *Fortran 90 Explained*. Oxford University Press, 1992.
9. A. Petrenko. Specification Based Testing: Towards Practice. In *Proceedings of "Perspectives of System Informatics"*, volume 2244 of *LNCS*, pages 287–300, 2001.
10. W. Grieskamp and Y. Gurevich and W. Schulte and M. Veanes. Testing with Abstract State Machines. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 257–261, February 2001.