Using ASM specification for automatic test suite generation for mpC parallel programming language compiler

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

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

Abstract. The paper presents an approach to automatic compiler test suite generation based on formal language specification. The language specification implemented using ASM formalism is discussed. The practical results for mpC parallel programming language compiler are presented. The advantages and drawbacks of proposed approach are discussed.

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 [9] 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 technique is also applicable to other parts of the language.

Our approach is a sort of "specification-based testing" [11, 12, 2]. We use Abstract State Machines [6] formalism for modeling mpC expressions semantics. The formal specification is implemented using the ASM-based Montages framework [8] – a new method for giving the semantics of a programming language.

We use the specification for three different purposes:

- **Generating test cases.** 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.
- **Generating test oracle.** 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 generation process. Practical results and future work are discussed in sections 4 and 5 respectively.

2 The ASM Specification for mpC Expressions

2.1 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[]) [5] supports array-based computations in the spirit of FORTRAN 90 [10]. The language introduces special operators for manipulating arrays as a whole.

The **grid** operator is used for addressing array sections. It has the following syntax:

```
expr[l : r : s]
```

where `expr` is an expression of an array or a pointer type and expressions `l`, `r`, `s` are expressions of integral type. The operands must satisfy the conditions: $l \leq r$, $l \leq 0$, $r \leq 0$, and $s > 0$. The result of the expression is a vector (an ordered sequence of objects) v containing $(r - l)/s + 1$ elements, with the value of the i-th element of the vector v being the value of the expression $expr[l + s * i]$. If $e[l + s * i]$ is an address expression, then v_i denotes the same object in the memory as $expr[l + s * i]$. Two or more grid operators applied consequently address sections of multidimensional arrays.

In C[] language unary and binary operators admit vector operands. In this case the operator is applied elementally to vector operands. For instance the following code computes the sum of arrays `a` and `b` elements and stores the result in array `c`:

```
int a[N], b[N], c[N];

...

c[0 : N - 1] = a[0 : N - 1] + b[0 : N - 1];
```

Another feature of C[] is **reduction operators**. The binary operators `+`, `*`, `|`, `&`, `^`, `||`, `&&`, `?>`, and `?<` have corresponding reduction counterparts: `[+]`, `[*]`,

`[|]`, `[&]`, `[^]`, `[||]`, `[&&]`. If `op` is a binary operator admitting operands of type T then corresponding reduction operator `[op]` is applicable to an expression of type "vector of elements of type T". The value of the expression `[op] expr` equals to the value of the expression `(...(v1 op v2) ... vn)`, where `v` is a n-element vector value of `expr` and `vi` denotes its i-th elements.

The following code gives an example of calculating dot product of two vectors:

```
double a[N], b[N], c;
...

c = [+](a[0 : N - 1 : 1] * b[0 : N - 1 : 1]);
```

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 [9] or [1]. For simplicity, examples in this paper use only vector expressions. However the proposed technique is implemented for all kinds of mpC expressions.

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.

2.2 Abstract State Machines

Abstract State Machine (ASM) is a new and powerful approach to specification of large-scale realistic software and hardware systems. We refer the reader to [6] for a detailed definition.

The state of an Abstract State Machine is given by the collection of functions on an abstract set called **superuniverse**. The basic ASM operation is an **update** which is defined as a function value modification at a given location (set of arguments):

$$f(t_1, \ldots, t_r) := t_{r+1}$$

The ASM is driven by transition rules. The expression above called an **update instruction** is a basic transition rule. More complex transition rules are obtained by recursive application of **sequence** and **conditional** constructors.

Sequence constructor The sequence of rules is a rule. The execution of a sequence of rules is defined as a simultaneous execution of rules comprising the sequence (i.e. all updates defined by the rules take place simultaneously).

Conditional constructor If $g_1, \ldots, g_k$ are Boolean terms and $R_1, \ldots, R_k$ are rules then the following expression is a rule:

$$
\begin{aligned}
&if\ g_1\ then\ R_1 \\
&elseif\ g_2\ then\ R_2 \\
&\vdots \\
&elseif\ g_k\ then\ R_k \\
&endif
\end{aligned}
$$

If at a given state S guard g_i holds and every g_j with $j < i$ fails then the execution of the rule described above is defined as the execution of the rule R_i.

The pure ASM constructs described above provides sufficient basis for the specification of any system. However in practice pure ASM specification may appear to be too cumbersome. That is why a number of ASM extensions have been introduced. One of those extensions is the XASM [3] language. It enriches ASM with several constructs providing more convenient way for specification of different aspects of the system behavior.

The XASM **do-forall** construct:

$$
\begin{aligned}
&do\ forall\ i\ in\ domain \\
&\ R \\
&enddo
\end{aligned}
$$

executes the rule R for all i from *domain* in parallel. This facility is extremely useful for giving the semantics of data-parallel language constructs such as binary operators and assignments of vector operands in mpC.

2.3 Montages.

Montages [8] are a semi-visual formalism for describing programming language syntax, static and dynamic semantics. Montages have been successfully used for the specification SQL [4], C [7] 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 rule, condition, dynamic semantics rules and a control-flow graph. Condition, static and dynamic semantics rules are written in XASM. Sample Montage for assignment operator is demonstrated on Fig. 1.

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

2.4 The Specification of mpC Expressions

The specification for mpC expressions consists of more then 30 Montages for mpC declarations and operators. In addition to *Value* and *Addr* attributes which are

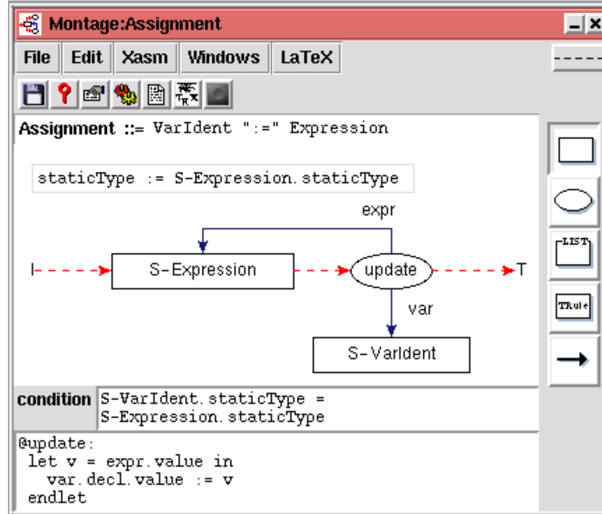

Fig. 1. Montage for assignment operator.

typically used for describing C expressions semantics two new – $VecValue$ and $VecAddr$ are introduced. Those attributes hold the vector value elements and addresses of vector elements respectively.

The XASM code portion below defines the dynamic semantics of a binary operator for the case of vector operands of the same size:

$$do\ forall\ i\ in\ set\ \{0..left.Type.Size - 1\}$$
$$self.VecValue(i) = ApplyBinaryOper(S - BinaryOper.Sign,$$
$$left.VecValue(i), right.VecValue(i))$$
$$enddo$$

In this example elements of the expression vector value are assigned to the result of applying the binary operator to corresponding elements of operands vector values.

The semantics of mpC expressions is given for AST nodes. The input language for executable specification is a text form of mpC AST representation. The fig. 2 demonstrates mpC expression and corresponding abstract syntax tree in graphical and text forms.

3 Generating the Test Suite from the Specification

The Test Suite Architecture. The test suite consists of mpC programs accompanied 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.

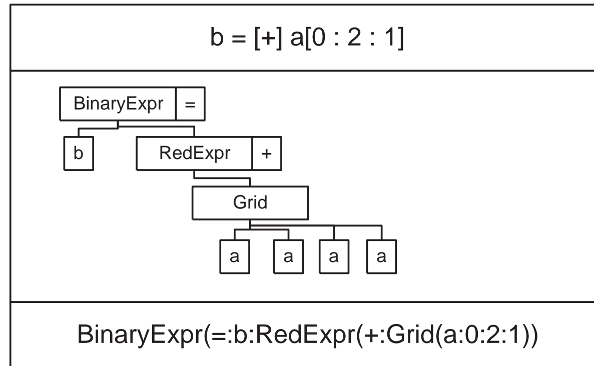

Fig. 2. mpC expression and its AST in graphical and text forms.

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 produced by the specification. If one of the mentioned steps fails the verdict is failure.

The Test Cases Generation Scheme. The proposed scheme of test suite generation is depicted at Fig. 3. We omit some technical details in order to make explanation clear and concise.

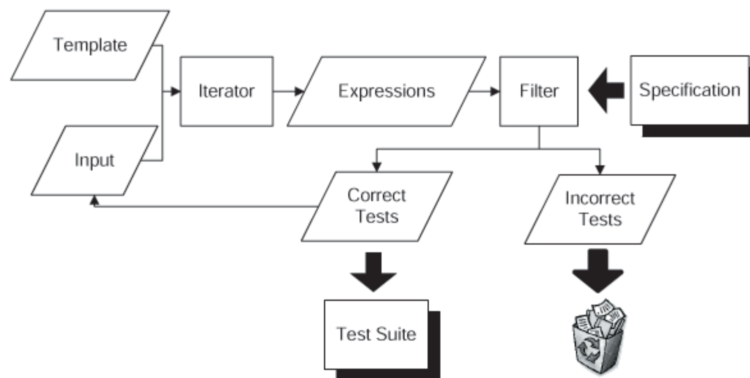

Fig. 3. The scheme of the test suite generation.

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 adding the initialization and output parts to it. 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.

We incorporate the coverage tracking into the Filter. It provides the possibility to track the coverage and a possibility to adjust filtering criteria upon coverage.

For the moment we use an update rule coverage [2] 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.

Obtaining the Test Case. Once the expression is generated we need several steps further to obtain a test case (see Fig. 4).

The first step is the **normalization**. The problem is that the generated expression may have an arbitrary type. If the expression has an arithmetic type the comparison with the trustable output is simple. In either case the normalization may be more difficult. For example if the expression has pointer type the straightforward comparison of values is senseless. The utility called **normalizer** applies several mpC operators to the generated expression in order to obtain the expression of the arithmetic type.

The test program is constructed by accomplishing the normalized expression with necessary declarations and initializations. The test program is processed by the executable specification to obtain the trustable output.

The final step in constructing the test case is the restoration of mpC source code from the AST test program. It is performed by the utility called **restorer**.

Example. Here we present a sample run of the generating scheme. Consider following Input and Template files:

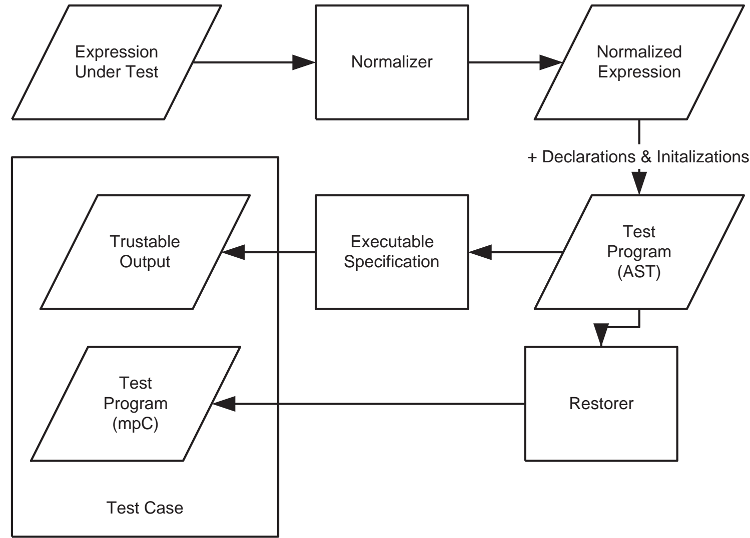

Fig. 4. Constructing the test case.

File	*Contents*	*Semantics*
Input	`s` `Grid(I:0:2:1)`	the variable of structure type and the expression `I[0:2:1]`
Template	`BinaryExpr(+ : $1 , $2)`	the binary operator

Tokens \$1 and \$2 denotes positions for substituting strings from the Template file. Having 2 entries in Input file and 2 places for substituting in Template file the Iterator produces 4 combinations:

```
BinaryExpr(+ : s : s),
BinaryExpr(+ : s : Grid(I:0:2:1)),
BinaryExpr(+ : Grid(I:0:2:1) : s),
BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1)).
```

Since the "+" operator doesn't admit operators of structure type the only semantically valid expression is `BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1))`, three remaining expressions are filtered out.

The first iteration produces one test case based on `BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1))` and the second iteration Input file consists of three entries:

```
s,
Grid(I:0:2:1),
BinaryExpr(+ : Grid(I:0:2:1) : Grid(I:0:2:1)).
```

The listing below demonstrates the resulting mpC test program obtained at the first iteration:

```
#include<stdio.h>
```

```
main(){
typedef int tInt;
typedef tInt*:(1) tPointer;
typedef struct {
tInt f;
tInt g;
} tStruct;
typedef tInt tArrInt[(3):(1)];
typedef tPointer tArrPointer[(3):(1)];
typedef tStruct tArrStruct[(3):(1)];
typedef tArrInt tArrArrInt[(3):(1)];
tInt i;
tPointer p;
tStruct s;
tArrInt I;
tArrPointer P;
tArrStruct S;
tArrArrInt II;
tInt Result;
(i)=(1);
(p)=(I);
((s).f)=(1);
((s).g)=(1);
((I)[(0):(2):(1)])=(i);
((P)[(0):(2):(1)])=(p);
((S)[(0):(2):(1)])=(s);
(((II)[(0):(2):(1)])[(0):(2):(1)])=(i);
printf("%d\n",([+](((I)[(0):(2):(1)])+((I)[(0):(2):(1)]))));
}
```

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 generation 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.

The advantage of the proposed approach is that the test suite generator is obtained for the price of almost nothing. The formal specification is a useful thing itself: the specification for mpC expressions discovered a lot of inconsistent places in the language specification as well as bugs in the compiler. In the test generating process we reuse the formal specification three times: for oracle, for filtering and for coverage tracking. The only thing we developed specifically for test generator is a set of simple scripts implementing the scheme depicted at Fig. 3.

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 generation 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 generation 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. 3).

Acknowledgments. We would like to thank Philipp Kutter and Mathias Anlauff for excellent assistance with Gem-Mex tool and Dr. A. Petrenko who inspired this work in ISPRAS.

References

1. www.ispras.ru/∼ mpc.

2. 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.
3. 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.
4. B. DiFranco. Specification of ISO SQL using Montages. Master's thesis, Università di l'Aquila, 1997.
5. Sergey Gaissaryan and Alexey Lastovetsky. ANSI C superset for vector and superscalar computers and its retargetable compiler. *Journal of C Language Translation*, 5, 1994.
6. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
7. J. Huggins and W. Shen. The Static and Dynamic Semantics of C. Technical Report CPSC-2000-4, Kettering University, Computer Science Program, 2000.
8. P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
9. 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.
10. M. Metcalf and J. Ried. *Fortran 90 Explained*. Oxford University Press, 1992.
11. A. Petrenko. Specification Based Testing: Towards Practice. In *Proceedings of "Perspectives of System Informatics"*, volume 2244 of *LNCS*, pages 287–300, 2001.
12. 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.