Applying Model Based Testing in Different Contexts*

Victor V. Kuliamin, Alexander K. Petrenko

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Kommunisticheskaya, 25, Moscow, Russia
{kuliamin,petrenko}@ispras.ru
http://www.ispras.ru/groups/rv/rv.html

Abstract. We describe ISP RAS experience in applications of model based testing in various areas. The two different examples are considered - UniTesK test development technology aimed at software component testing and OTK tool intended to be used in test development for complex structured text processors, the main example of which is compilers. The surprising fact is that the two methods used in the tools have different prerequisites for successful applications in industrial software development. This demonstrates possibility to change those prerequisites by changing the technical aspects of the method applied. Both techniques were developed in RedVerst group of ISP RAS [1].
Keywords: software component testing, compiler testing, testing based on software contracts, testing based on finite automata, test data generation

1 Introduction

Any researcher when trying to apply the methods and tools he designed in real industrial projects faces with the fact that the success of such an application is determined mostly by the context of the project and not by the technical characteristics of the method or tool themselves. At least three aspects of the project context can be distinguished and all of them have significant influence on the project results.

- Project staff, that is personal capabilities, skills, and education of the project participants.
- Organizational issues, that is the structure and policies of the organization where the project is conducted. This point also includes the processes and technologies already used in the organization, the management style used, and so on.
- General social issues, including approaches and paradigms of design and development dominating in the current period and in the considered industry branch, national and cultural specifics of country and region, and other issues those are not specific to one organization.

* This work is partially supported by RFBR grant 02-01-00959, by grant of Russian Science Support Foundation, and by Program 4 of Mathematics Branch of RAS.

All these aspects should be seriously considered when we trying to make a successful application. Each advanced technique born in the research community and model based testing technique as well has its own prerequisites to the project context, which when hold make the probability of success much higher.

But we may ask ourselves a question, whether it is possible to modify the technical aspects of the technology in so far that its prerequisites to the project context are eased. Our experience shows that it is possible for model based testing, at least sometimes.

This article provides a description of two model based testing techniques developed in ISP RAS and used in various industrial projects. The first technique, UniTesK, is similar to classic model based testing approaches and is intended for use in software component testing. The second one, OTK, possesses some uncommon features and is used for compiler testing. The next two sections present details of the techniques. The conclusion summarizes the article and provides some ideas on future development of the techniques.

2 UniTesK Approach

UniTesK [2–5] test development technology was designed on the base of experience of ISP RAS joint project with Nortel Networks, conducted in 1994–1996 [6]. The goal of the project was to develop a regression test suite for the telecommunication switch operating system kernel. The project was finished successfully and during the validation of the resulting test suite it was applied to the existing version of switch OS and detected a lot of bugs in it, although this product was actively used in the field for several years.

UniTesK is a model based test development technology for general purpose software components. The solutions provided by the technology can be presented as follows.

- Functionality of the software components under test is described as *software contracts*. A component contract consists of preconditions and postconditions of its operations, a model of its state (if those operations are state-dependent), and invariants stating data integrity constraints on the state fields.
- Test adequacy criteria used are based on coverage of specifications obtained. Operations' postconditions are partitioned into parts called *functionality branches* and describing areas of 'homogeneous' operation behavior. 'Homogeneous' hear means that the operation's behavior in different such areas is described by different sets of constraints. Test adequacy is measured as a quotient of covered functionality branches to the total number of them.
- To construct an executable test targeted at high coverage level achievement *test scenario* is used. Test scenario is a kind of test script, which contains a generic description of (input-output) finite state machine, the transition tour on which provides the coverage level needed. States of this state machine correspond to large groups of states of the component model.Input symbols

```
namespace Chase.Examples {
  specification public class AccountSpecification {
    public static int maximumCredit = 3;
    public int balance = 0;

    invariant I( "Balance is not less than -maxCredit" ) {
      return balance >= -maximumCredit;
    }

    specification public void Deposit( int sum )
      reads sum
      updates balance
    {
      pre {
        return (0 < sum ) && (balance <= System.Int32.MaxValue - sum);
      }
      post {
        branch Single( "Single branch" );
        return balance == pre balance + sum;
      }
    }

    specification public int Withdraw( int sum )
      reads sum, maximumCredit
      updates balance
    {
      pre { return sum > 0; }
      post {
        if( balance < sum - maximumCredit ) {
          branch TooLargeSum( "Withdrawn sum is too large" );
          return balance == pre balance && $this.Result == 0;
        } else {
          branch NormalSum( "Successful withdrawal" );
          return balance == pre balance - sum && $this.Result == sum;
        }
      }
    }
  }
}
```

Fig. 1. Example of specifications in C# extension

correspond to functionality branches of the operations under test. Test scenario doesn't usually define some specific state machine, but gives a generic description of a large set of machines. Namely, it defines what is considered as states and what input symbols can be applied in a state. The specific finite state machine is determined on execution stage only and depends on parameters of test execution and the implementation under test.

- To bind model with the implementation under test special adapters (historically called *mediators* in UniTesk) are used.
- Test generation is performed on-the-fly. After start of a test scenario *the test engine* implementing traversal algorithm for some class of finite state machines generates a sequence of inputs to cover all reachable transitions.

Test development techniques used in UniTesK have a lot of similarities with the ones used in AsmL Tester [7, 8], Spec# [9], and Gotcha [10, 11] tools. The main difference with these tools is use of explicitly written test scenarios representing the model used for test sequence generation and factorization technique

for constructing a test scenario on the base of contract specifications and coverage criterion [12].

```
namespace Chase.Examples {
  scenario public class AccountScenario {
    public static void Main() {
      Tracer.Init();
      AccountScenario test = new AccountScenario( 13 );
      test.Run();
      Tracer.Finish();
    }

    AccountSpecification target = new AccountSpecification();
    int maxBalance = 10;

    protected virtual void configureMediators() {
      target = mediator AccountMediator( new Account() );
      target.AttachOracle();
    }

    public AccountScenario( int maxBalance ) {
      Engine = new Chase.Engines.DFSWithSCEngine();
      this.maxBalance = maxBalance;
      configureMediators();
    }

    public override Chase.Lang.ModelObject State {
      get { return new Chase.States.IntState( target.balance ); }
    }
    scenario Deposit() {
      if( target.balance < maxBalance )
        iterate( int i = 1; i < 4; i++; ) target.Deposit(i);
      return true;
    }

    scenario Withdraw() {
      iterate(int i = 1; i < 6; i++; ) target.Withdraw(i);
      return true;
    }
  }
}
```

Fig. 2. Example of test scenario in C# extension

To simplify the usage and mastering of the technology software contracts, adapters and test scenarios are developed in extensions of widely used programming languages. UniTesK technology is supported now by several tools:

- J@T [13] based on Java extension and integrated in NetBeans development environment and Sun ONE Studio. Java extension called J@tva was developed in 2000 [14], the first version of tool was developed in 2000–2001.
- Ch@se [15] based on C# extension and integrated in Microsoft Visual Studio .NET 2003. C# extension was developed in 2002, the tool – in 2003.
- CTesK [16] based on C extension and integrated in Microsoft Visual Studio 6.0. Both the language extension and the tool were developed in 2001–2002.

Example of contract specifications in C# extension for a class implementing banking account with credit possibility is presented on Fig. 1. Test scenario and adapter for the same example are presented on Fig. 2 and Fig. 3 correspondingly.

```
namespace Chase.Examples {
  mediator public class AccountMediator : AccountSpecification {
    implementation public Account target = null;

    static AccountMediator() {
      maximumCredit = Account.maximumCredit;
    }

    mediator public void Deposit( int sum ) {
      implementation { target.Deposit( sum ); }
    }

    mediator public int Withdraw( int sum ) {
      implementation { return target.Withdraw( sum ); }
    }

    public override void MapStateUp() {
      balance = target.balance;
    }
  }
}
```

Fig. 3. Example of adapter in C# extension

In Appendix you can find figures with pages of reports generated from results of test execution on the account implementation with erroneous output in `Withdraw()` method when the withdrawn sum is too big. Fig. 5 presents the achieved test coverage of `Withdraw()` method in terms of functionality barnches and information on number of failures detected. Fig. 6 shows the MCS diagram of a part of test execution.

UniTesK tools and techniques were used in several case studies, including test development for several different implementations of IPv6 protocol, banking client data management software, enterprise software framework, windows services, etc. The complete list of case studies can be found on [5].

The experience of applying UniTesK tools in various projects conducted by different organizations shows, that the following conditions are necessary for the application project to be successful.

– The test development team should have access to knowledge source on actual functionality of the system under test. This source can be represented as the source code of the system, its actual project documentation, requirements specification documents. In case of ambiguity or incompleteness of documentation (which is always the case in real life projects) or instead of using documentation at all test developers should have direct access to developers, architects, business analysts, or the experts who know the detailed functionality of the system. Incomplete, vague, outdated documentation, absence of communication with experts can dramatically decrease the performance of UniTesK test development and make it useless at all.
– Tests should have direct access to the interface of the system under test. If the component under test is included in bigger system, which interface we should use to test the embedded component, then the functionality of this system should be considered in details. This usually increases the complexity

of test development and introduces additional risks related with access to knowledge on the bigger system.

3 OTK Approach

OTK [17, 18] tool and the underlying test construction technique were developed during joint project of ISP and Intel on testing a set of optimizer units of Intel C++ and Fortran compilers in 2001–2003.

In this project both prerequisites for UniTesK application success were broken.

- ISP has no access to knowledge sources on actual functionality of optimizers due to Intel security policy – no access to the source code, no communication with developers. The only information available was that an optimization algorithm operates similar to the one described in certain section of the Muchnick's book [19].
- ISP has no access to the interface of optimizer units, which are working with internal representation of program code. We had to deal with the external compiler interface only, that is to write test programs, to compile them and hope that optimizer is activated during this operation.

Nevertheless, the results of the project are quite promising – the optimizer testing technique and supporting tool were developed, and several bugs found. The technique developed seems to be perspective for use in various formal text processor testing.

The main solutions used in the test development technique can be summarized as follows.

- The role of *test data model* is played by generic program structure described in terms of syntactic elements and abstracted from the elements irrelevant for the processing algorithm we need to test. Such a model can be obtained by analysis of the algorithm and determining the structural elements of programs and their connections, which the algorithm deals with.
- *Test program generator* is constructed as a structured system of generators of separate syntactic elements. Such generators in their turn are constructed from generators of subelements, and so on. For example, generator of binary expressions is usually constructed form two generators of subexpressions and generator of operation signs. All these generators work with model representation of program structure. The text of test programs appears after applying special *mapper* component transforming model representation into textual and constructed also on the base of syntactic structure.
- Test adequacy criteria are based on coverage of algorithm provided by the generator constructed. The latter is controlled mostly by heuristics used in test program generator constructions and generator's parameters.
- Checking optimizer correctness is organized as comparison of traces generated by program compiled with optimization and without it. This mechanism

can be applied to program transformation units preserving its semantics, but except for optimizers examples of practically significant such transformation units are very rare.

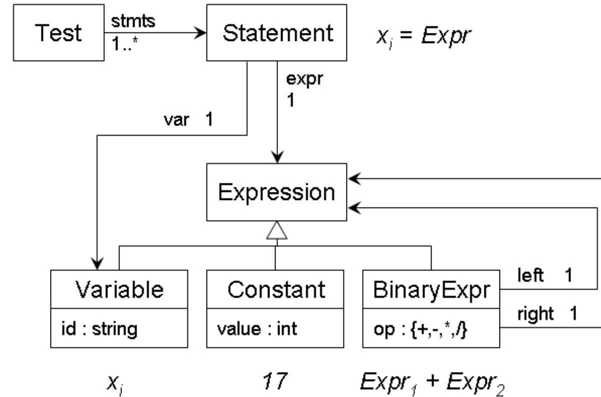

Fig. 4. Syntactic elements relevant for CSE optimization

Let us consider common subexpression elimination (CSE) optimization. Fig. 4 presents the UML-like model of program elements significant for this algorithm. They include expressions and their parts (for simplicity we consider here only basic unary and binary expressions). Fig. 7 in Appendix demonstrates the textual representation of that diagram actually used by OTK tool.

Fig. 8 in Appendix shows an example of a test program generated with OTK from that model. The program consists of several statements, each of which is assignment having some binary expression on its right side. The program takes several parameters that are used to initialize local variables, which are modified by the assignments and then are printed in the trace. Traces of optimized and nonoptimized programs' executions with several arrays of parameters are compared to find differences in their behavior. Each difference detected is further analyzed for being caused by a bug in an optimizer unit.

4 Conclusion

The UniTesK approach for model based testing has shown itself as practically applicable in industrial software development (see [5] for description of case studies in different domains). Nevertheless, its poses certain restrictions of the project where we want to use it really effectively.

OTK tool and the underlying test construction techniques can be used in less restricted environment, but they appears to be specific for formal text processors.

Possible further development of UniTesK and OTK is the following.

– Integration of OTK technique in UniTesK tools as a method of test data generation in case of complex input data.
– Extension of OTK approach into the model based method of compiler testing. Works in this direction is conducted by ISP now and the following steps can be distinguished.
  • Testing of syntax checker units. This is rather simple task, various solutions for which already suggested in research society.
  • Testing of static semantics checker units. ISP is currently working on possible solutions for this task. Some results can be found in [20, 21], although are not central in these articles.
  • Testing of compiler back-end. This task seems to be the most complex, since some formal definition of language semantics is needed to solve it. Works [20, 21] concerns mostly this aspect, but deals with only a small part of language.

Very tempting is to suggest model based testing techniques that also have simplified restriction on the environment where they can be applied and it seems to be possible. Such techniques may be domain specific (like OTK) and not as general as the ones developed so far.

References

1. http://www.ispras.ru/groups/rv/rv.html
2. V. Kuliamin, A. Petrenko, I. Bourdonov, and A. Kossatchev. UniTesK Test Suite Architecture. Proc. of FME 2002, LNCS 2391, pp. 77–88, Springer-Verlag, 2002.
3. V. Kuliamin, A. Petrenko, A. Kossatchev, and I. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. In proceedings of 1-st Europpean Conference on Model-Driven Software Engineering, December 2003.
4. V. Kuliamin, A. Petrenko, N. Pakoulin, I. Bourdonov, and A. Kossatchev. Integration of Functional and Timed Testing of Real-time and Concurrent Systems. Proc. of PSI 2003, LNCS 2890, pp. 450–461, Springer-Verlag, 2003.
5. http://www.unitesk.com
6. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS 1708, Springer-Verlag, 1999, pp. 608–621.
7. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Testing with Abstract State Machines. In R. Moreno-Diaz and A. Quesada-Arencibia, eds., Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001), Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain, February 2001, pp. 257–261.
8. http://research.microsoft.com/fse/asml/
9. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: an Overview. To appear in Proceedings of CASSIS International Workshop, Marseille, 2004.
10. E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. IBM Systems Journal, volume 41, Number 1, 2002, pp. 89–110.
11. http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html

12. I. Burdonov, A. Kossatchev, and V. Kulyamin. Application of finite automatons for program testing. Programming and Computer Software, 26(2):61–73, 2000. (Translation from Russian).
13. http://www.unitesk.com/papers/jat/jat14_gs.pdf
14. V. Kuliamin, A. Petrenko, I. Bourdonov, A. Demakov, A. Jarov, A. Kossatchev, and S. Zelenov. Java Specification Extension for Automated Test Development. Proceedings of PSI'01. LNCS 2244, pp. 301–307. Springer-Verlag, 2001.
15. http://www.unitesk.com/papers/chase/chase_gs.pdf
16. http://www.unitesk.com/papers/ctesk/ctesk_gs.pdf
17. S. Zelenov, S. Zelenova, A. Kossatchev, A. Petrenko. Test Generation for Compilers and Other Formal Text Processors. Programming and Computer Software, Vol. 29, No. 2, pp. 104-111, 2003. (Translation from Russian)
18. A. Kossatchev, A. Petrenko, S. Zelenov, S. Zelenova. Using Model-Based Approach for Automated Testing of Optimizing Compilers. Proc. Intl. Workshop on Program Undestanding, Gorno-Altaisk, 2003.
19. Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, August 1997.
20. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, V. Shishkov. Using ASM Specifications for Compiler Testing. Proc. of Abstract State Machines 2003: 415.
21. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, V. Shishkov. Coverage-driven Automated Compiler Test Suite Generation. Electr. Notes Theor. Comput. Sci. 82(3): 2003.

5 Appendix

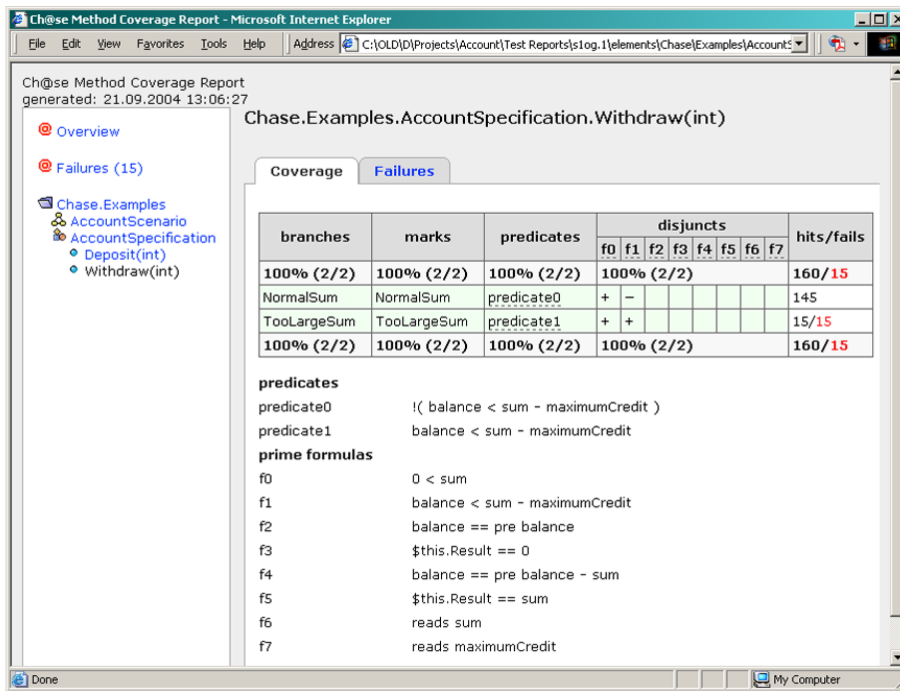

Fig. 5. Coverage of `Withdraw()` method with failures

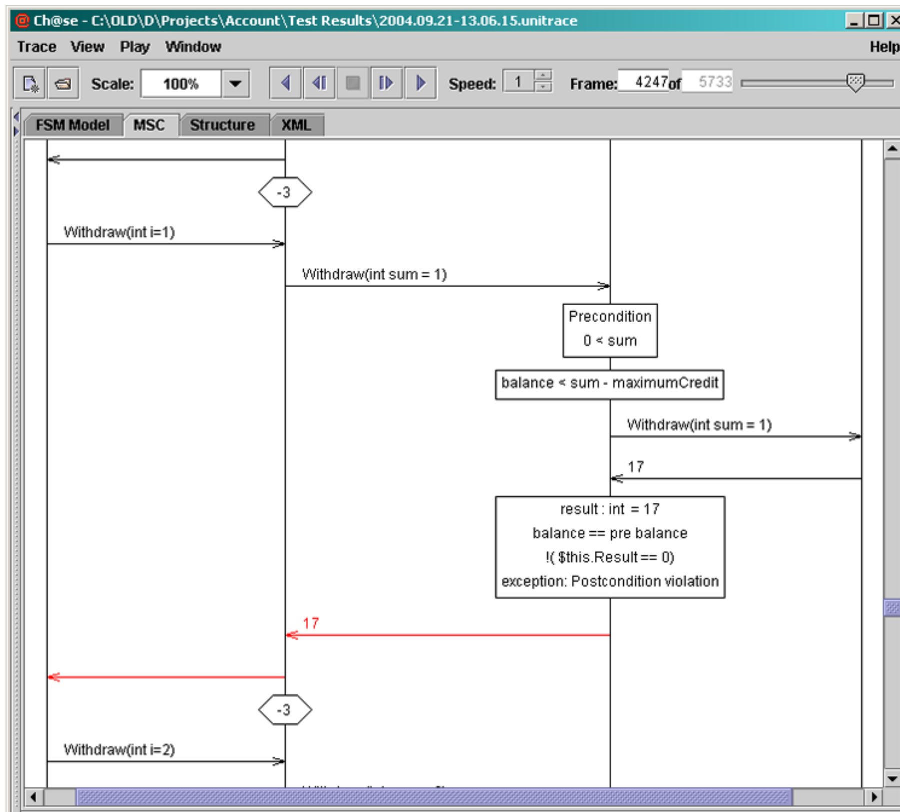

Fig. 6. MSC diagram of test execution

```
tree root.Model;

header { import ru.ispras.redverst.optest.OtkNode; }

node Test : <OtkNode> {
  child Stmt* stmts;
}

node Stmt : <OtkNode> {
  child Expr expr;
  child Var var;
}

abstract node Expr : <OtkNode> { }

node BinaryExpr : Expr {
  child Expr left;
  child Expr right;
  attribute <int> op;
}

node Var : Expr {
  attribute <int> id;
}

node Constant : Expr {
  attribute <int> value;
}
```

Fig. 7. Textual representation of CSE-related program model

```
void test_113 ( int v0, int v1, int v2, int v3 ) {
  int x0 = v0;
  int x1 = v1;
  int x2 = v2;
  int x3 = v3;

  x0 = ( ( x0 * x0 ) * ( x3 + x2 ) );
  x1 = ( ( x0 * x0 ) * ( x3 + x3 ) );
  x2 = ( ( x0 * x0 ) * ( x3 - 1 ) );
  x3 = ( ( x0 * x0 ) * ( x3 - 2 ) );
  x0 = ( ( x0 * x0 ) * ( x3 - 3 ) );
  x1 = ( ( x0 * x0 ) * ( x3 - x0 ) );
  x2 = ( ( x0 * x0 ) * ( x3 - x1 ) );
  x3 = ( ( x0 * x0 ) * ( x3 - x2 ) );
  x0 = ( ( x0 * x0 ) * ( x3 - x3 ) );
  x1 = ( ( x0 * x0 ) * ( x3 * 1 ) );
  x2 = ( ( x0 * x0 ) * ( x3 * 2 ) );
  x3 = ( ( x0 * x0 ) * ( x3 * 3 ) );
  x0 = ( ( x0 * x0 ) * ( x3 * x0 ) );
  x1 = ( ( x0 * x0 ) * ( x3 * x1 ) );
  x2 = ( ( x0 * x0 ) * ( x3 * x2 ) );
  x3 = ( ( x0 * x0 ) * ( x3 * x3 ) );
  x0 = ( ( x0 * x1 ) + ( x0 + 1 ) );
  x1 = ( ( x0 * x1 ) + ( x0 + 2 ) );
  x2 = ( ( x0 * x1 ) + ( x0 + 3 ) );
  x3 = ( ( x0 * x1 ) + ( x0 + x0 ) );
  x0 = ( ( x0 * x1 ) + ( x0 + x1 ) );
  x1 = ( ( x0 * x1 ) + ( x0 + x2 ) );
  x2 = ( ( x0 * x1 ) + ( x0 + x3 ) );
  x3 = ( ( x0 * x1 ) + ( x0 - 1 ) );
  x0 = ( ( x0 * x1 ) + ( x0 - 2 ) );
  x1 = ( ( x0 * x1 ) + ( x0 - 3 ) );
  x2 = ( ( x0 * x1 ) + ( x0 - x0 ) );
  x3 = ( ( x0 * x1 ) + ( x0 - x1 ) );
  x0 = ( ( x0 * x1 ) + ( x0 - x2 ) );
  x1 = ( ( x0 * x1 ) + ( x0 - x3 ) );
  x2 = ( ( x0 * x1 ) + ( x0 * 1 ) );
  x3 = ( ( x0 * x1 ) + ( x0 * 2 ) );
  x0 = ( ( x0 * x1 ) + ( x0 * 3 ) );
  x1 = ( ( x0 * x1 ) + ( x0 * x0 ) );
  x2 = ( ( x0 * x1 ) + ( x0 * x1 ) );
  x3 = ( ( x0 * x1 ) + ( x0 * x2 ) );
  x0 = ( ( x0 * x1 ) + ( x0 * x3 ) );
  x1 = ( ( x0 * x1 ) + ( x1 + 1 ) );
  x2 = ( ( x0 * x1 ) + ( x1 + 2 ) );
  x3 = ( ( x0 * x1 ) + ( x1 + 3 ) );

  printf("%d, %d, %d, %d", x0, x1, x2, x3);
}
```

Fig. 8. Example of generated test program